

# 深度学习 原理与实践

THE PRINCIPLE AND PRACTICE OF DEEP LEARNING

陈仲铭 彭凌西◎著



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS



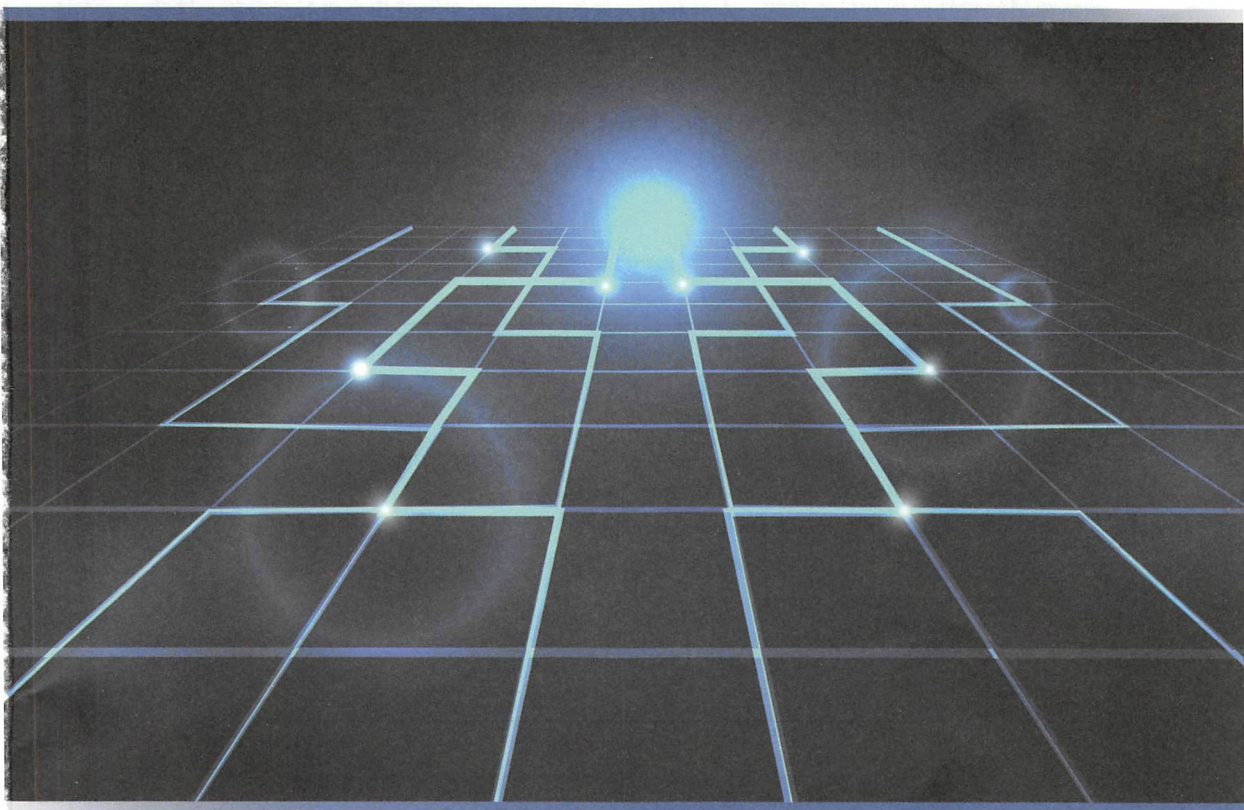
版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

# 深度学习 原理与实践

THE PRINCIPLE AND PRACTICE OF DEEP LEARNING

陈仲铭 彭凌西◎著



人民邮电出版社  
北京



## 图书在版编目 (C I P) 数据

深度学习原理与实践 / 陈仲铭, 彭凌西著. -- 北京:  
人民邮电出版社, 2018. 8  
ISBN 978-7-115-48367-6

I. ①深… II. ①陈… ②彭… III. ①机器学习—研  
究 IV. ①TP181

中国版本图书馆CIP数据核字(2018)第112818号

## 内 容 提 要

本书详细介绍了目前深度学习相关的常用网络模型 (ANN、CNN、RNN), 以及不同网络模型的算法原理和核心思想。本书利用大量的实例代码对网络模型进行了分析, 这些案例能够加深读者对网络模型的认识。此外, 本书还提供完整的进阶内容和对应案例, 让读者全面深入地了解深度学习的知识和技巧, 达到学以致用目的。

本书适用于大数据平台系统工程师、算法工程师、数据科学家, 可作为对人工智能和深度学习感兴趣的计算机相关从业人员的学习用书, 也可作为计算机等相关专业的师生用书和培训学校的教材。

- 
- ◆ 著 陈仲铭 彭凌西  
责任编辑 张 爽  
责任印制 焦志炜
  - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
北京瑞禾彩色印刷有限公司印刷
  - ◆ 开本: 720×960 1/16  
印张: 21.25  
字数: 392 千字 2018 年 8 月第 1 版  
印数: 1—3 000 册 2018 年 8 月北京第 1 次印刷
- 

定价: 89.00 元

读者服务热线: (010) 81055410 印装质量热线: (010) 81055316

反盗版热线: (010) 81055315

广告经营许可证: 京东工商广登字 20170147 号

# 序

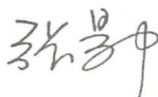
人工智能领域一直备受国内外著名研究专家和学者的关注。近年来，以机器学习、深度学习、神经网络及模拟计算为代表的计算智能技术得到空前发展，掀起了人工智能领域的新高潮。

这其中特别值得一提的是深度学习。科学家已经证明，深度学习技术的深度表征能力能够有效地提取数据的高维特征。深度学习模型从简单的特征开始，通过神经网络逐层组合的方式，不断地抽取更加复杂的特征到下一层，最终提取到高维的抽象特征。深度学习模型具有特殊的网络结构组织方式，使网络层数不断加深，加上特殊模型训练技巧，反向梯度算法和随机梯度算法的组合，在理论上可以表达任何函数的网络模型。深度学习因为具有的大规模并行分布式处理、自组织、自学习、自适应、可迁移学习的能力，以及数据多样性，对序列信号有记忆功能和鲁棒性等特点，因此受到国内外众多学者的高度重视。

受深度学习这一技术的影响，近年来涌现出了 ImageNet、COCO、Pascal、VOT 等与深度学习相关的全球算法竞赛，腾讯、阿里巴巴、Google、Facebook、Amazon 等国内外著名互联网企业相继建立各自的人工智能实验室。仅 2017 年，国际上举办有关深度学习的著名专题会议高达 28 场，其中包括 ICCV、ICRL、NIPS 等。

基于深度学习与生物神经网络的相似性，近年来研究者们提出了深度神经网络这一新概念。在计算机推荐系统、图像感知、模拟学习等领域，传统机器学习的弱点是其表征能力受限。深度学习的及时出现，为众多应用领域带来新的挑战，具有广阔的应用前景。

“深度学习”的相关图书近年来层出不穷，但是将原理阐述和实践紧密结合的书籍相对较少。本书对深度学习的基本概念、原理及应用技术做了深入浅出的讲解，它的出版对从事人工智能、机器学习、深度学习研究的科技工作者和研究生大有裨益。相信本书的出版会促进国内深度学习的研究和应用，有助于我国新一代人工智能创新活动的蓬勃发展。



中国科学院院士

2018 年 1 月 10 日于广州



# 前言

随着深度学习的爆发，截至目前，最新的深度学习模型已经远远超越了传统的机器学习算法，在数据的预测准确性和分类精度上都达到了前所未有的高度。深度学习的表征功能不需要工程师手动提取数据中的隐含特征，而是自动地深度挖掘数据并提取数据中隐含的高维特征。可以说深度学习实实在在地节省了工程师们在特征工程上的时间，提高了算法的精度和准确率，也推动着实际工程的进步。

深度学习的出现为机器学习和人工智能领域带来了新的机会和希望，极大地拓展了人们对于计算人工智能的想象力和应用范畴，使很多的计算辅助功能成为可能。深度学习已经不仅仅是计算机科学领域的问题，它结合了神经科学和逻辑学科的知识，涉及众多跨学科领域的知识交叉。从图像感知、无人驾驶汽车、无人驾驶飞机，到生物医学的预防性诊断、病理预测，甚至是更加贴近年轻一代的电影推荐、购物指南，几乎所有领域都可以使用深度学习。而在某些领域中，人类所掌握的知识还是如此地微不足道，因此深度学习在这些领域仍有巨大的发展空间。

在深度学习领域最权威的书是 Ian Goodfellow、Yoshua Bengio 和 Aaron Courville 编写的 *Deep Learning*（中文版由人民邮电出版社出版）。但与深度学习相关，且将知识原理与案例实践相结合的书籍在市场上并不多见。作者写作本书的初衷就是帮助更多的人了解深度学习，并投身于人工智能领域。

一年多以来，作者业余时间几乎都用于写作、编写示例代码，也曾经想过放弃，但都坚持了下来。成功不就是一次次坚持而成就的吗？人生，多多少少也有些梦想，值得我们去付出！

## 学习建议

*Conquer yourself rather than the world.*

征服你自己，而不是去征服世界。

——笛卡儿《谈谈方法》

我们努力提高自己的能力、学习深度学习和人工智能，并不是为了与别人一决高下，而是超越自己。成功的路上会有很多建议，而针对本书的学习，有下面几条建议。

## 2 | 前言

### 1. 学习 Python 基础知识

贯穿全书中的代码由 Python 编程语言编写而成，但是本书并没有对 Python 代码进行过多的解释，而是围绕和专注于讲解深度学习的原理和思想。因此希望读者先自行安装 Python 3，了解 Python 编程语言的特性和基本使用方法，有一定的了解后即可直接使用本书的示例代码，这对于理解深度学习会有所帮助。

### 2. 实践是检验真理的唯一标准

每一行代码和函数都是作者亲自实现过的，尽管不同服务器和不同版本的框架会存在一些差异，也可能读者看到本书代码时已经稍显过时，但我仍然希望读者亲自去尝试实现书中的代码，毕竟“纸上得来终觉浅”，实践才是检验真理的唯一标准。

### 3. 不要纠结于框架

深度学习好比“菜谱”，数据就是“肉和青菜”，深度学习框架就是“炒菜的锅和铲”。谁说红烧土豆一定要用“Caffe”牌的锅，香煎莲藕一定要用“Google”牌的铲？我们只要学会菜谱，研究“锅”和“铲”的属性并多加实践练习，就可以炒出一盘香喷喷的菜。希望读者不要纠结于本书所用的是 Keras 框架，还是 Tensorflow 框架，而是将所有的目光都聚焦在深度学习的原理和案例实践上。

### 4. 多阅读相关文献

读杰出的书籍，有如和杰出的人物促膝交谈。深度学习的知识日新月异，在知识更新迭代迅速的时代，我们需要掌握知识的本质内容，而掌握本质内容的最好方法之一就是阅读与知识点相关的论文文献，去理解与思考为什么要这样，这样做的优缺点是什么。多阅读相关的文献，我们就会更好地把握住深度学习知识的本质。

## 本书特色

(1) 大量图例，简单易懂。作者亲自绘制了大量插图，力求还原深度学习的算法思想，分解和剖析晦涩的算法，用图例来表示复杂的问题。生动的图例也能给读者带来阅读乐趣，快乐地学习算法知识，体会深度学习的算法本质。

(2) 简化公式，生动比喻。深度学习和机器学习类的书中通常会有大量复杂冗长的算法公式，为了避免出现读者读不懂的情况，本书尽可能地统一了公式和符号，简化相关公式，并加以生动的比喻进行解析。在启发读者的同时，锻炼读者分析问题和解决问题的能力。

(3) 算法原理，代码实现。在介绍深度学习及相关算法的原理时，不仅给出了对应的公式，还给出了实现和求解公式的代码，让读者明确该算法的作用、输入和输出。原理与代码相结合，使得读者对深度学习的算法实现更加具有亲切感。

(4) 深入浅出，精心剖析。理解深度学习需要一定的机器学习知识，本书在第 1 章介绍了深度学习与机器学习的关系，并简要介绍了机器学习的内容。在内容安排



上，每章依次介绍模型框架的应用场景、结构和使用方式，最后通过真实的案例去全面分析该模型结构。目的是让读者可以抓住深度学习的本质。

(5) 入门实践，案例重现。每一章最后的真实案例不是直接堆砌代码，而是讲解使用该算法模型的原因和好处。从简单的背景知识出发，使用前文讲解过的深度学习知识实现一个实际的工程项目。实践可以用于及时检验读者对所学知识的掌握程度，为读者奠定深度学习的实践基础。

## 建议和反馈

为了让广大读者更好地理解和使用书中的案例代码，本书提供了一个公开的 GitHub 代码库：<https://github.com/chenzomi12/deeplearning-inaction/>。

完成一本书的编著与出版是一项极其琐碎和繁重的工作，我已尽力完善本书内容，但仍然可能存在纰漏和错误之处，敬请各位读者朋友指正，请致信作者邮箱 [chenzomi12@gmail.com](mailto:chenzomi12@gmail.com) 或本书编辑邮箱 [zhangshuang@ptpress.com.cn](mailto:zhangshuang@ptpress.com.cn)。

作者衷心地希望各位读者能够从本书获益，这是对我最大的支持和鼓励。

## 致谢

感谢每一位为此书做出贡献的朋友。

感谢每一位读者，希望本书的内容值得您投入宝贵的时间。

感谢王佳博士在本书目录和内容结构上提出的建议。

感谢人民邮电出版社的张爽编辑，谢谢您的精心策划和建议。

感谢广东海格通讯有限公司的领导吕韶清和星舆科技的领导古明晖，让我在工作中有机会接触机器视觉、机器学习和深度学习，这是我开始撰写本书的契机。

感谢广东工业大学的研究生刘尚武、中山大学的段永强两位兄弟对本书的大力支持，他们对本书进行了多次的审阅和批注，并提出了宝贵的意见，本书的每一章内容都经过他们两人的精心修改。正是他们的付出，才有了这本通俗易懂的深度学习读物。

感谢我尊敬的父母、亲爱的姐姐陈泳茵以及爱人欧阳素行，在高度紧张的工作氛围和高强度加班的环境下，在我业余时间给予我大力的支持和鼓励，让我有勇气和耐心完成一次又一次的改动和编辑，并在写作的语言细节上给了我很多启发。

最后，感谢国家自然科学基金资助（编号 61100150 和 61772147）、广东省高校创新团队项目资助（编号 2015KCXTD014），以及广东省高等学校自然科学研究重大项目资助（编号 2014KZDXM044）。

陈仲铭

2018 年 1 月

# 目 录

第1章 初探深度学习 .....	1
1.1 什么是深度学习 .....	2
1.1.1 机器学习的一般方法 .....	4
1.1.2 选择深度学习的原因 .....	5
1.1.3 深度学习前的思考 .....	6
1.2 深度学习的应用 .....	7
1.3 深度学习的硬件加速器 .....	10
1.3.1 GPU比CPU更适合深度学习 .....	10
1.3.2 GPU硬件选择 .....	13
1.4 深度学习的软件框架 .....	15
1.5 本章小结 .....	19
引用/参考 .....	19
第2章 人工神经网络 .....	21
2.1 人工神经网络概述 .....	22
2.1.1 历史背景 .....	22
2.1.2 基本单位——神经元 .....	24
2.1.3 线性模型与激活函数 .....	25
2.1.4 多层神经网络 .....	26
2.2 训练与预测 .....	28
2.2.1 训练 .....	28
2.2.2 预测 .....	29
2.3 核心算法 .....	29
2.3.1 梯度下降算法 .....	29
2.3.2 向前传播算法 .....	31



## 2 | 目 录

2.3.3	反向传播算法	33
2.4	示例：医疗数据诊断	42
2.4.1	从医疗数据到数学模型	43
2.4.2	准备数据	44
2.4.3	线性分类	45
2.4.4	建立人工神经网络模型	46
2.4.5	隐层节点数对人工神经网络模型的影响	53
2.5	本章小结	55
	引用/参考	56
第3章	深度学习基础及技巧	58
3.1	激活函数	59
3.1.1	线性函数	60
3.1.2	Sigmoid函数	61
3.1.3	双曲正切函数	62
3.1.4	ReLU函数	63
3.1.5	Softmax函数	64
3.1.6	激活函数的选择	65
3.2	损失函数	65
3.2.1	损失函数的定义	66
3.2.2	回归损失函数	67
3.2.3	分类损失函数	69
3.2.4	神经网络中常用的损失函数	70
3.3	超参数	71
3.3.1	学习率	71
3.3.2	动量	72
3.4	深度学习的技巧	73
3.4.1	数据集准备	73
3.4.2	数据集扩展	74
3.4.3	数据预处理	76
3.4.4	网络的初始化	81
3.4.5	网络过度拟合	84
3.4.6	正则化方法	85

3.4.7 GPU的使用 .....	88
3.4.8 训练过程的技巧 .....	89
3.5 本章小结 .....	91
引用/参考 .....	92
<b>第4章 卷积神经网络 .....</b>	<b>93</b>
4.1 卷积神经网络概述 .....	94
4.1.1 卷积神经网络的应用 .....	95
4.1.2 卷积神经网络的结构 .....	99
4.1.3 卷积神经网络与人工神经网络的联系 .....	101
4.2 卷积操作 .....	103
4.2.1 滑动窗口卷积操作 .....	104
4.2.2 网络卷积层操作 .....	105
4.2.3 矩阵快速卷积 .....	107
4.3 卷积神经网络三大核心思想 .....	110
4.3.1 传统神经网络的缺点 .....	110
4.3.2 局部感知 .....	111
4.3.3 权值共享 .....	112
4.3.4 下采样 .....	113
4.4 设计卷积神经网络架构 .....	115
4.4.1 网络层间排列规律 .....	116
4.4.2 网络参数设计规律 .....	116
4.5 示例1: 可视化手写字体网络特征 .....	117
4.5.1 MNIST手写字体数据库 .....	118
4.5.2 LeNet5网络模型 .....	119
4.5.3 LeNet5网络训练 .....	122
4.5.4 可视化特征向量 .....	124
4.6 示例2: 少样本卷积神经网络分类 .....	127
4.6.1 Kaggle猫狗数据库 .....	128
4.6.2 AlexNet模型 .....	130
4.6.3 AlexNet训练 .....	134
4.6.4 AlexNet预测 .....	136
4.6.5 微调网络 .....	137

4.7 本章小结 .....	140
引用/参考 .....	141
第5章 卷积神经网络视觉盛宴 .....	143
5.1 图像目标检测 .....	144
5.1.1 传统目标检测方法 .....	146
5.1.2 基于区域的网络 .....	147
5.1.3 基于回归的网络 .....	157
5.1.4 目标检测小结 .....	163
5.2 图像语义分割 .....	165
5.2.1 传统图像分割方法 .....	165
5.2.2 全卷积神经网络 .....	167
5.2.3 SegNet网络 .....	171
5.2.4 DeepLab网络 .....	173
5.3 示例1: NMS确定候选框 .....	176
5.4 示例2: SS算法提取候选框 .....	179
5.4.1 图像复杂度 .....	179
5.4.2 算法核心思想 .....	180
5.4.3 区域相似度计算 .....	184
5.5 本章小结 .....	189
引用/参考 .....	190
第6章 卷积神经网络进阶示例 .....	192
6.1 示例1: 全卷积网络图像语义分割 .....	193
6.1.1 VGG连续小核卷积层 .....	193
6.1.2 VGG网络模型 .....	195
6.1.3 全卷积网络模型 .....	199
6.1.4 全卷积网络语义分割 .....	204
6.2 示例2: 深度可视化网络 .....	209
6.2.1 梯度上升法 .....	210
6.2.2 可视化所有卷积层 .....	213
6.2.3 可视化输出层 .....	218
6.2.4 卷积神经网络真能理解视觉吗 .....	219



6.3	示例3: 卷积神经网络艺术绘画 .....	220
6.3.1	算法思想 .....	221
6.3.2	图像风格定义 .....	222
6.3.3	图像内容定义 .....	224
6.3.4	算法实现 .....	225
	引用/参考 .....	229
第7章	循环神经网络 .....	231
7.1	初识循环神经网络 .....	232
7.1.1	前馈式神经网络的缺点 .....	233
7.1.2	什么是序列数据 .....	234
7.2	循环神经网络的应用 .....	235
7.3	循环神经网络的模型结构 .....	237
7.3.1	序列数据建模 .....	237
7.3.2	基本结构 .....	238
7.3.3	其他结构 .....	239
7.4	循环神经网络的核心算法 .....	241
7.4.1	模型详解 .....	241
7.4.2	向前传播 .....	243
7.4.3	损失函数 .....	245
7.4.4	时间反向传播算法 .....	246
7.4.5	梯度消失与梯度爆炸 .....	251
7.5	示例: 使用循环神经网络预测文本数据 .....	254
7.5.1	定义网络模型 .....	254
7.5.2	序列数据预处理 .....	255
7.5.3	准备输入输出数据 .....	258
7.5.4	实现简单的循环神经网络模型 .....	260
7.6	本章小结 .....	263
	引用/参考 .....	264
第8章	循环神经网络进阶序列长期记忆 .....	265
8.1	长期依赖问题 .....	266
8.2	长短期记忆网络 .....	269

8.2.1	LSTM网络结构 .....	269
8.2.2	LSTM记忆单元 .....	270
8.2.3	LSTM记忆方式 .....	274
8.3	门控循环单元 .....	274
8.3.1	GRU记忆单元 .....	275
8.3.2	GRU实现 .....	276
8.3.3	GRU与LSTM比较 .....	277
8.4	示例1: 神奇的机器翻译 .....	278
8.4.1	基于统计的机器翻译 .....	279
8.4.2	基于神经网络的机器翻译 .....	282
8.4.3	编码-解码模型 .....	283
8.4.4	平衡语料数据集 .....	287
8.4.5	机器翻译的未来 .....	292
8.5	示例2: 智能对话机器人 .....	293
8.5.1	Seq2Seq模型 .....	294
8.5.2	Seq2Seq模型的缺点 .....	295
8.5.3	超越Seq2Seq框架 .....	296
8.6	示例3: 智能语音识别音箱 .....	299
8.6.1	语音识别框架 .....	300
8.6.2	准备语音数据 .....	302
8.6.3	语音特征提取 .....	306
8.6.4	声学模型 .....	311
8.6.5	语言模型 .....	323
8.6.6	语音识别的展望 .....	323
8.7	本章小结 .....	324
	引用/参考 .....	325

---

# 第 1 章

---

## 初探深度学习

本章主要内容：

- 了解什么是深度学习
- 了解深度学习的加速硬件
- 了解深度学习的软件框架



我们暂且不管深度学习是什么，深度学习有多强大。作为信息行业的杰出的工程师，首先需要知道深度学习真正带来的是什么？未来，深度学习对社会以及各个行业会带来什么影响？拥有大量深度学习人才的中国企业将会在上充当何种角色？深度学习又会给我们带来什么样的机遇与挑战？

“路漫漫其修远兮”。深度学习需要大量的数据和庞大的计算资源，而这就将我们的视线带入 GPU 的世界。如果时至今日，你还以为英伟达（NVIDIA）只是一家卖显卡的公司，那就显得有点孤陋寡闻了。因为如今的特斯拉自动驾驶系统 Autopilot 2.0、行车预警系统、无人采集系统、智能物流系统等，到处都充斥着 NVIDIA 的身影。正是因为深度学习，NVIDIA 才会成为新一轮人工智能公司中的独角兽。

深度学习在各个领域带来的变革才刚刚开始，如黎明破晓一样让人心潮澎湃。近期关于无人驾驶、辅助驾驶、智能音箱的新闻越来越多，无论大公司，还是新创公司都在谈人工智能，为什么呢？因为深度学习极大地降低了技术的门槛。只要拥有充足的数据，加点硬件成本和时间成本，就可以利用深度学习这一技术实现各种方案，这是新创公司实现弯道超车的机会。

在本章中，我们将会了解到什么是深度学习，探索深度学习的应用，知晓深度学习的强大。近年来深度学习呈爆发式增长，主要得益于其相关硬件加速器和软件平台的迅猛发展。因此本章将会讲解深度学习最常用的硬件平台，及其对应的软件架构平台。通过对深度学习初步的了解，相信上面的问题将会给我们带来更多关于技术的思考。

## 1.1 什么是深度学习

2016 年年初，由 Google DeepMind 开发的 AlphaGo 在围棋大战中打败了韩国的围棋大师李世石。就如同 1997 年 IBM 的“深蓝”计算机战胜了世界首席国际象棋大师卡斯帕罗夫一样，媒体开始铺天盖地般地宣传人工智能时代的来临。同时，资本开始追捧与人工智能产业相关的公司，一时间收购并购人工智能企业的狂潮席卷而来。

在描述 DeepMind 胜利的时候，很多媒体都会把人工智能（Artificial Intelligence）、机器学习（Machine Learning）和深度学习（Deep Learning）混为一谈。虽然从严格定义上来说，DeepMind 在 AlphaGo 程序中对上述 3 种技术都有所使用，但其真正使用更多的是深度学习。

下面来了解人工智能、机器学习、深度学习这三者之间的关系。如图 1-1 所示，人工智能包含着机器学习，而深度学习则是机器学习的重要分支之一，它们三者是包含与被包含的关系。

从 20 世纪 50 年代，人工智能第一次提出至今，人工智能的问题基本上已经定



- 具有更多的神经元；
- 具有更复杂的网络连接方式；
- 拥有惊人的计算量；
- 能够自动提取数据高维特征。

本文介绍的深度网络主要是指具有深层的神经网络，包括：人工神经网络（ANN）、卷积神经网络（CNN）、循环神经网络（RNN）。

### 1.1.1 机器学习的一般方法

机器学习按照方法主要可以分为两大类：监督学习和无监督学习。其中监督学习主要由分类和回归等问题组成，无监督学习主要由聚类 and 关联分析等问题组成。深度学习则属于监督学习当中的一种。

机器学习中的监督学习是指使用算法对结构化或者非结构化的有标注的数据进行解析，从数据中学习，获取数据中特定的结构模型，然后使用这些结构或者模型来对未知的新数据进行预测。通俗来说，监督学习就是通过对数据进行分析，找到数据的表达模型，有了这个模型就可以对新输入的数据套用该模型来做决策。

图 1-3 为监督学习的一般方法，主要分为训练和预测阶段。在训练阶段（对数据进行分析的阶段），首先需要根据原始的数据进行特征提取，这个过程叫作特征工程。得到特征后，我们可以使用决策树、随机森林等模型算法去分析数据之间的特征或者关系，最终得到关于输入数据的模型（Model）。在预测阶段，同样按照特征工程的方法提取了数据后，使用训练阶段得到的模型对特征向量进行预测，最终得到所属的标签（Labels）。

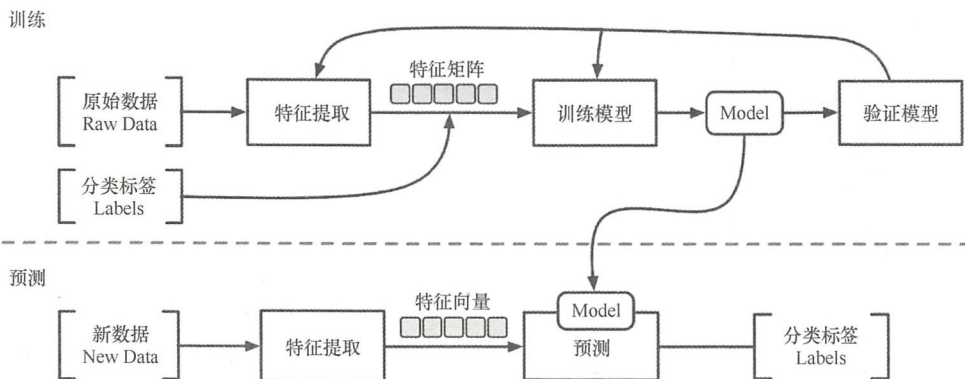


图 1-3 监督学习一般方法，分为训练阶段和预测阶段，训练阶段的目标是得到训练模型，而预测阶段的目标是使用训练模型对输入数据进行预测



机器学习算法中的每一种不同的模型都有其自身的规则去解释输入的数据，然后对新输入的数据进行预测和判断。例如决策树的模型，就是构建一个树形结构，每一个节点代表一种数据的类型，每一个叶子节点则代表一种类别。线性回归模型则利用线性回归方程创建一组参数来表示输入的数据之间的关系，而神经网络则有一组权重参数向量来代表节点之间的关系。

到目前为止，我们有各种各样的机器学习算法可以对数据进行建模。通过算法对数据进行建模后，我们可以学习到数据之间的关联性，提取数据的高维特征，并且训练完的新模型可以对相类似的新数据进行预测。可现实情况真的会这么理想吗？图 1-3 中对应的每一个步骤所花费的时间是相同的吗？

机器学习领域当中最出彩的莫过于计算机视觉，这里我们暂且抛开计算机视觉的图像获取需要通过硬件编码而获取彩色或者灰度图像等一系列硬件的工作。举一个例子，如果我们想要识别道路上的停车牌，首先不是直接对图片进行滑动窗口，而是先进行边缘检测、中值滤波、高斯滤波等图像预处理操作，待得到工程师们认为可以使用的图像之后，才开始对图像进行分析，产生一系列的物体候选框。然后通过图像检测的算法对各个窗口进行特征检测，通过机器学习的算法提取每个窗口的特征，或者可以通过数据挖掘的算法对数据进行压缩，从而提取到更加抽象的特征数据信息。接着我们需要定义分类器，对停车牌中的字母进行判断，例如可以使用简单的 SVM 分类器对“STOP”4 个字母的抽象信息进行分类。到此为止，算是能够使用机器学习的算法去感知图像，识别输入的图像是否为停车牌。

虽然上面的图像分类结果基本可用，但是还远远达不到全自动化识别的目的。当遇到雨雪天气时，马路一旁的标志牌已经不是那么清晰可见，抑或标志牌被树木遮挡时，标志牌的特征也会改变。这时传统机器学习算法就不是那么有效了，这也是一直困扰学者们的一大难题，因此各种旋转不变性、光照不变性算法相继被提出，但始终没有某一种算法特别奏效。

随着时间的推移，深度学习迅速地改变了这一切。

### 1.1.2 选择深度学习的原因

在深度学习出现之前，机器学习的工程师们往往需要花费数天、数周，甚至数月的时间去收集数据，然后对数据进行筛选，尝试各种不同的特征提取方法对数据进行提取，或者结合几种不同的特征对数据进行分类和预测。作者曾经使用传统机器学习检测数据时，就花费了大量的时间尝试使用各种不同的算法、特征提取方式和分类器，对数据进行特征提取。经过大量的尝试，才找到满足业务要求的特定方

法模型。随着深度学习的爆发，最新的深度学习算法已经远远超越了传统的机器学习算法对于数据的预测和分类精度。深度学习不需要我们自己去提取特征，而是自动地对数据进行筛选，自动地提取数据高维特征。深度学习的一般方法（见图 1-4）与传统机器学习中的监督学习一般方法（见图 1-3）相比，少了特征工程，节约了工程师们大量工作时间。

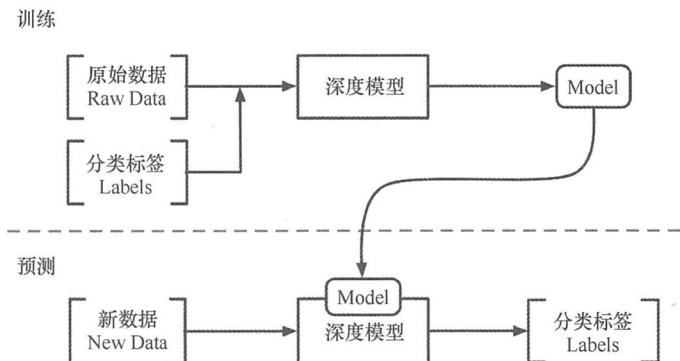


图 1-4 深度学习一般方法

选择深度学习，一方面因为它节省工程师们的时间，降低工程师的工作量，提高工作效率，从而让工程师们把更多的时间和精力投入在更有价值的研究方向上；另一方面是深度学习的效果，已经在众多领域开始超越传统的机器学习算法，甚至在某些领域能够获得比人类预测的更好的效果；此外，深度学习还可以与大数据无缝结合，输入庞大的数据集进行大数据端到端的学习过程，这种大道至简的理念吸引着无数的研究者。

深度学习已经不仅仅是计算机科学领域的问题，它结合了更多关于神经网络的问题，涉及生物学、神经科学等众多领域，仍有非常巨大的发展空间。深度学习就像一座宝矿，我们看到的或许只是冰山一角……

### 1.1.3 深度学习前的思考

听上去显得有点高端大气，但想要正确开启深度学习之门，最好的方法是先回答下面 4 个问题：

- 需要给深度网络模型输入什么样的数据？
- 想要从深度网络模型中提取什么样类型的数据？
- 选择哪种深度网络模型更加适用于手头上的数据？

- 根据这个深度网络模型我们希望从新的数据得到一个什么样的结果？

第一个问题是给模型输入什么样的数据，回答这个问题的时候我们就大概知道应该选择什么样的数据作为样本，样本的形式是图片、文档，还是语音。明确了要从模型中提取什么类型的数据体后，我们就会更加清晰地定义网络模型的损失函数。第三个问题可以让我们进一步理清脉络，将问题落实到神经网络的细节中，例如卷积神经网络 CNN 中大概是用多少层的网络，循环神经网络 RNN 中应该定义多少层循环和时间步等。最后一个问题将更加有利于我们把深度学习的算法与模型结合到工程项目当中，真正帮助我们解决实际问题。

如果我们能够很好地回答上述这 4 个问题，表明我们对所遇到的问题有了充分的理解和分析，也就能够针对特定的任务（数据和场景）去寻找到合适的工作流程或者工作方式，快速建立属于自己的深度学习模型了！

## 1.2 深度学习的应用

1990 年左右，人工神经网络的理论已经取得重要的突破，那时很多新发表的论文和计算机架构上都是基于神经网络。神经网络应用的突破领域之一是控制论，神经网络有着一套完美的反馈机制，给控制论增添了不少色彩。而深度学习的出现就如寒武纪生命大爆发一样，前几年我们或许听到更多的是大数据处理、数据挖掘，而如今在科技创新的生态中，几乎每个人都在谈论深度学习、人工智能。下面简单来介绍关于深度学习的应用。

### 1. 图像处理

YouTube 每天大概上传 6 亿个视频，Facebook 每天上传超过 180 亿张图片。全球每天产生的图像和视频的数据量大得惊人，科技公司每天都在为了这些资源的存储和标注耗费着大量的服务器和人力资源。

以图片为例，为了更好地对图片进行存储，我们需要记录下与图像有关联的信息，其中每张图片在系统内部或外部都需要进行标注，编写与图像相关的摘要。如图 1-5 所示，通过深度学习技术可以获取图像中相关物体的信息，对图像进行语义分割，还可以从图像生成摘要。如果用户偷懒或者忘记填写相关信息，深度学习可以帮助用户进行信息补充，给予充分的信息提示功能。对于图片数据库后台系统管理人员来说，深度学习可以帮助数据库管理员填写与真实图片相关的内容，以便于数据后续的利用与备份。这无疑为科技企业带来更多有效的数据可以去展开数据分析和数据挖掘，从而更好地去建立用户画像。



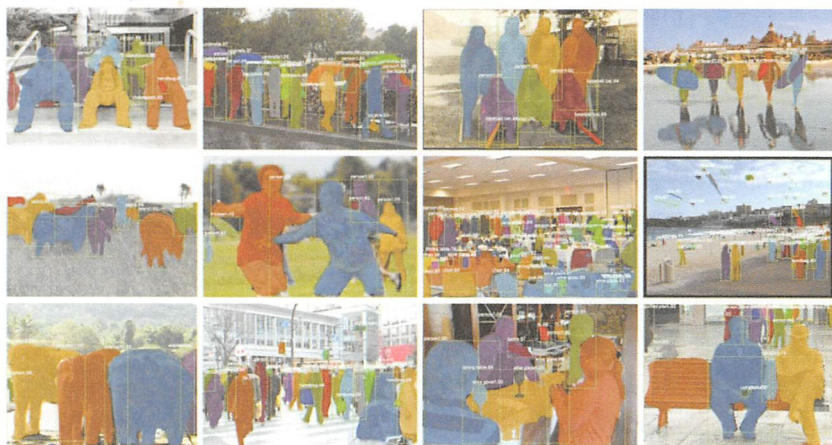


图 1-5 Mask R-CNN 对图像进行目标检测和图形语义分割

## 2. 自动驾驶与高精度地图

自动驾驶作为感知和控制技术的制高点，会在未来几年发生巨大的进步。目前业界成熟的感知系统所能够完成的功能已经完全不能满足自动驾驶的需求，而基于深度学习的感知算法能够区别于传统的感知，是深度学习助力自动驾驶的核心所在。使用深度学习可以检测出路面，精确地识别出不同的光照、场景、时间、地点、形状的车前物体以及障碍。我们不仅可以检测出行人、路面上的障碍物、行驶标志牌等，还可以利用这些信息去制作高精度地图，为自动驾驶控制提供底层信息驱动。深度学习技术将进一步解决计算机感知的问题，使自动驾驶汽车向量产的方向迈进了一大步（见图 1-6）。

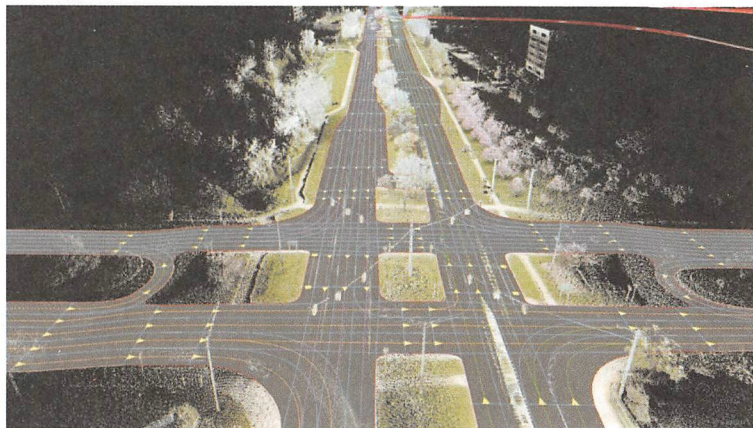


图 1-6 高精度地图局部示例图。通过使用深度学习的感知算法对激光雷达和摄像头采集到的路面信息进行融合，制作成高精度地图



### 3. 机器人

深度学习技术的突破使得机器人的复杂感知变为可能。Google 使用深度神经网络训练机械臂根据摄像头的输入和电机的传动命令抓取物体，同时机器人会根据当前机械臂的状态及时纠正其位置从而便于抓取目标物体。为了加快学习进程，Google 还同时使用多个机械臂进行将近 3000 小时的训练。在约 80 万次的抓取尝试后，开始看到智能反应行为，其机械臂抓取物体的错误率大大降低。

### 4. 医疗健康诊断

2011 年，IBM 机器人 Watson 开始利用深度学习技术对医学知识进行学习和研究。经过了 4 年多的训练，在学习了 200 本肿瘤领域的教科书、290 种医学期刊和超过 1500 万份的文献后，Watson 开始被应用在临床上，在肺癌、乳腺癌、直肠癌、结肠癌、胃癌和宫颈癌等领域向人类医生提出建议。

另外，医疗领域深度学习团队 Airdoc 目前已经掌握了世界领先的图像识别能力。结合数学、医学知识和深度学习算法后，在人类医学专家的帮助下，在心血管、肿瘤、神经内科、五官等领域建立了多个精准深度学习医学辅助诊断模型，并取得了良好的进展（见图 1-7 和图 1-8）。

总体来说，深度学习在医疗健康领域的机遇主要有 7 大方向：一是提供临床诊断辅助系统等医疗服务，应用于早期筛查、诊断、康复、手术风险评估场景；二是医疗机构的信息化，通过数据分析，帮助医疗机构提升运营效率；三是进行医学影像识别，帮助医生更快更准地读取病人的影像；四是利用医疗大数据，助力医疗机构大数据可视化及数据价值提升；五是在药企研发领域，解决药品研发周期长、成本高的问题；六是健康管理服务，通过包括可穿戴设备在内的手段，监测用户个人健康数据，预测和管控疾病风险；七是在基因测序领域，将深度学习用于分析基因数据，推进精准医疗。

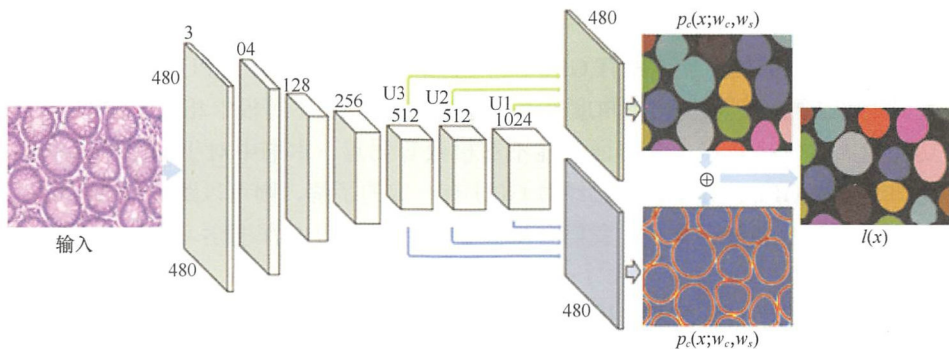


图 1-7 利用深度学习技术对细胞影像图进行分割，检查病变细胞

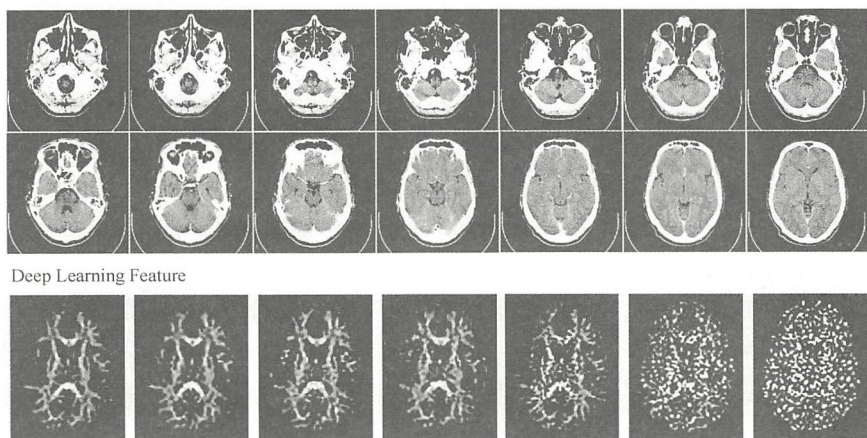


图 1-8 使用深度学习对核磁共振（NMRI）图进行特征提取

诸如上述的应用数不胜数，深度学习技术已经开始渗透到每一个领域当中，使得机器学习能够实现更多的应用场景，并且极大地拓展了人工智能的领域范畴。从无人驾驶汽车、无人驾驶飞机，到生物医学的预防性诊断、病理预测，甚至是更加贴近年轻一代的电影推荐、购物指南，几乎所有领域都可以使用深度学习。

## 1.3 深度学习的硬件加速器

硬件加速器的选择在深度学习中至为关键。没有加速硬件，无论算法多么复杂、模型能够处理多少数据量，也只是空谈。硬件加速器是深度学习应用的核心要素，能够获得巨大的计算性能提升。硬件加速器的作用之巨大不可忽视。随着在大数据下的并行计算和硬件加速器的高速发展，计算机的运算能力取得了飞跃式的前进，也让深度学习、人工智能迈入了一个新的技术时代。

传统意义上的硬件加速器有 GPU、FPGA 和 ASIC。其中，GPU 的优势在于性能强大、生态成熟，但从另一个角度来说，与 FPGA、ASIC 等板卡相比，GPU 也会遇到功耗较大、价格较贵、某方面性能不够极致等弱点。本书中对于深度学习默认的硬件加速器为 GPU，如果读者已经对 GPU 有一定的了解，则可以跳过本节。如果对 GPU 的了解较少也没有关系，我们会在本节中介绍 GPU 与深度学习的关系。

### 1.3.1 GPU比CPU更适合深度学习

GPU 作为硬件加速器之一，通过大量图形处理单元与 CPU 协同工作，对深度学

习、数据分析，以及大量计算的工程应用进行加速。从 2007 年 NVIDIA 公司发布了第一个支持 CUDA 的 GPU 后，GPU 的应用范围不断拓展，从政府实验室、大学、企业的大型数据中心，到现今非常火热的人工智能汽车、无人驾驶飞机和机器人等嵌入式平台，GPU 都发挥着巨大的作用。

## CUDA

统一计算设备架构（Compute Unified Device Architecture，CUDA）。随着显卡的发展，GPU 越来越强大，GPU 开始主要为显示图像做优化，在计算上已经超越了通用的 CPU。如此强大的芯片如果只是作为显卡就太浪费了，因此 NVIDIA 推出 CUDA 这一通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。

### 1. GPU 与 CPU 比较

比较 GPU 和 CPU，就是比较它们两者如何处理任务。如图 1-9 所示，CPU 使用几个核心处理单元去优化串行顺序任务，而 GPU 的大规模并行架构拥有数以千计的更小、更高效的处理单元，用于处理多个并行小任务。

既然 GPU 计算规模如此惊人，那为什么不用 GPU 代替 CPU 去工作呢？因为 GPU 的工作方式相对简单，不能完成复杂的逻辑工作，也缺少相应的指令集；而 CPU 拥有复杂的系统指令，能够进行复杂的任务操作和调度，两者是互补关系，而不能相互代替。

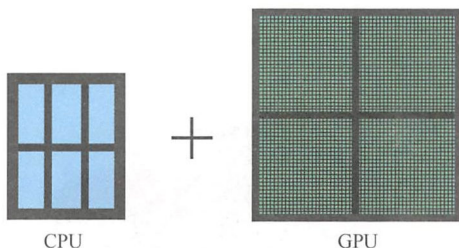


图 1-9 CPU 中只有 4 个核心并行工作，GPU 拥有 256 个核心并行工作

### 2. GPU 更适合深度学习

GPU 是大规模并行架构，处理并行任务毫无疑问是非常快的，深度学习需要高效的矩阵操作和大量的卷积操作，GPU 的并行架构再适合不过。简单来说，确实如此，但是为什么 GPU 进行矩阵操作和卷积操作会比 CPU 要快呢？真正原因是 GPU 具有如下特性：

- 高带宽；
- 高速的缓存性能；
- 并行单元多。

性能瓶颈通常不是因为芯片的数学计算的吞吐量，而是受芯片的内存带宽限制。由于在图形处理器 GPU 上包含了非常多的数学逻辑单元 ALU，因此有可能造成输入数据的速率无法维持如此高的计算速率，所以 GPU 需要高带宽。

在执行多任务时，CPU 需要等待带宽，而 GPU 能够优化带宽。举个简单的例



子，我们可以把 CPU 看作跑车，GPU 是大卡车，如图 1-10 所示任务就是要将一堆货物从北京搬运到广州。CPU（跑车）可以快速地把数据（货物）从内存读入 RAM 中，然而 GPU（大卡车）装货的速度就好慢了。不过后面才是重点，CPU（跑车）把这堆数据（货物）从北京搬运到广州需要来回操作很多次，也就是往返京广线很多次，而 GPU（大卡车）只需要一次就可以完成搬运（一次可以装载大量数据进入内存）。换言之，CPU 擅长操作小的内存块，而 GPU 则擅长操作大的内存块。CPU 集群大概可以达到 50GB/s 的带宽总量，而等量的 GPU 集群可以达到 750GB/s 的带宽量。

由于计算操作都是对缓存数据进行处理，所以缓存越大，其计算性能越高。但问题是，GPU（大卡车）一次能够加载很多数据进入缓存，但是由于 I/O 的下限决定传输性能，如果 GPU 装载一次货物时需要等待很长时间，那么缓存就算再大也没有意义。对于这个问题，GPU 是如何解决呢？



图 1-10 跑车和卡车

如果让一辆大卡车去装载很多堆货物，就要等待很长的时间了，因为要等待大卡车从北京运到广州，然后再回来装货物。设想一下，我们现在拥有了跑车车队和卡车车队（线程并行），运载一堆货物（非常大块的内存数据需要读入缓存，如大型矩阵）。我们会等待第一辆卡车，但是后面就不需要等待的时间了，因为在广州会有一队伍的大卡车正在排队输送货物（数据），这时处理器就可以直接从缓存中读取数据了。在线性并行的情况下，GPU 可以提供高带宽，从而隐藏延迟时间。这也就是 GPU 比 CPU 更适合处理深度学习的原因。

我们很少会看到豪华阵容的跑车车队在高速上载货，但是会经常看到货车车队在高速上载货。CPU 的线程再多，作用也是不大的，因此 Intel 推出的 CPU 最普遍的还是 2 核、4 核的 i 系列，核数过多对于 CPU 来说可能会造成浪费。

存储器快速地把大块的数据从 RAM 中读入缓存（见大卡车车队和跑车车队的例子）只是 GPU 的优势之一，第二个优势是当数据被读入寄存器中可以马上被大量并行的 GPU 核心处理器直接执行。



在 CPU 的 x86 架构中，有读取速度很快的 L1 高速缓存和内部寄存器，这些存储器件距离执行单元非常近，方便其对寄存器的数据进行快速调用。距离执行单元（CPU 核）越远，其访问存储器的响应时间越慢。例如地上有 2 张等额的人民币，我们的第一反应是先把距离最近的一张先捡起来，再去捡距离稍远一张。CPU 对于缓存的大小是有物理限制的，不是越大越好。那么 GPU 的缓存这么大是不是不好呢？

GPU 多核的优势就在这里体现了，对于每一个处理单元，GPU 会有很多对应的小寄存器。这样我们就可以有大量寄存器存储和执行数据，这些寄存器虽然很小，但是数据传输速度和执行速度非常快。举个例子，与 CPU 同样价格的 GPU，却拥有比 CPU 多出 30 倍数量的寄存器，14MB 的寄存器数可以处理速度高达 80TB/s 的数据。

这样将 CPU 与 GPU 的寄存器来对比，对于 CPU 来说是不公平的，毕竟 CPU 的功能和指令更加丰富，就像在跑车中我们可以听歌，享受豪车带来的喜悦。另外一方面，处理器的寄存器越多，就越难被充分利用。像对 GPU 的大量小寄存器进行操作，如果由我们自己来写编译指令，估计会让一块顶配的 Titan X 当成一块 8 位单片机来使用。不过 NVIDIA 开发出 NVCC 编译器去做这件事情，NVCC 能够充分利用 GPU 中的寄存器和 L1 高速缓存的性能，就解决了 GPU 多核大缓存的问题。

现在我们用 GPU 来进行深度学习，在 L1 高速缓存中和 GPU 的寄存器上存储大量的数据，反复使用卷积操作和矩阵乘法操作，而不用担心运算速度慢的问题。假设有一个 100MB 的矩阵，我们可以根据寄存器的数量和高速缓存的大小将该矩阵分解成多个如  $3 \times 3$  的小矩阵，然后以 10~80TB/s 的速度与一个三通道的  $3 \times 3$  的小矩阵相乘完成一次卷积操作。

以上 3 点：高带宽存储器；在线程并行下隐藏存储器访问延迟；大量而且快速的寄存器和 L1 高速缓存，使 GPU 比 CPU 更加适合用于深度学习的计算。

## 1.3.2 GPU 硬件选择

正如上一节所说，GPU 为深度学习带来了卓越的性能。在深度学习中使用 GPU 时，我们会经常为它带来的速度提升而惊叹：在线性的数学问题求解上的其速度能比 CPU 快了将近 5 倍，在一些更加复杂的并行操作上其速度能够快 50 倍之多。在 GPU 的帮助下，我们可以更快地试验新的想法、算法和实验，并迅速得到反馈，如验证哪些网络模型效果更好，哪些网络模型效果一般。如果资金充足，建议一定要使用 GPU。下面来看应该使用什么样的 GPU 进行深度学习。

### 1. AMD、NVIDIA、Xeon Phi 选择谁

对于深度学习的加速器 GPU，现在市面上主要的品牌有 AMD、NVIDIA、

Intel 的 Xeon Phi (见图 1-11)。其中 NVIDIA 的计算加速标准库 cuDNN 使得工程师在 CUDA 平台中构建深度学习变得非常容易, 而且在同一张显卡的前提下比没有使用 cuDNN 的速度提升 5 倍之多。此外, 从英伟达公司与特斯拉汽车公司合作而发布的车载超级计算机 Drive PX2 来看, 英伟达公司押注人工智能与深度学习, 使得该公司的 GPU 得到充分的资源支持。最后, 对于 CUDA 平台来说, 其社区已经趋于完善, 很多开源解决方案为后续编程提供了可靠的建议。相对而言, AMD 的显卡对于深度学习的支持稍微落后, 现阶段并没有非常合适的深度学习库, 并且社区并非十分完善。

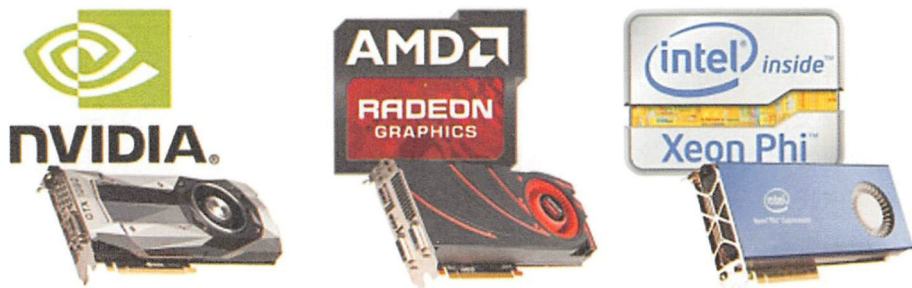


图 1-11 AMD、NVIDIA、Xeon Phi 显卡对比

对于 Intel 公司的 GPU, 根据 Tim Dettmers 的博客, 他曾经研究过超过 500 个 Xeon Phi 集群, 遭遇了很多的挫折。原因有:

- Xeon Phi 的数学核心函数库并不兼容 NumPy;
- Xeon Phi 编译器不支持 C++ 11 的一些特性;
- 处理器的调度算法有待优化。

如果操作的张量 tensor 尺寸连续变化, 线程调度器 (thread scheduler) 中的漏洞或问题会削弱其计算性能。例如, 如果有不同大小的全连接层或 dropout 层, Xeon Phi 会比 CPU 还慢。最后, Tim Dettmers 给出的建议是: 想学习深度学习, 请远离 Xeon Phi! 虽然该博客文章可能有些过时, 但是建议读者在研究深度学习之前选择一个社区较为活跃的平台, 尽量少走弯路。

图 1-12 所示为某公司的深度学习硬件平台, 使用 4 个 GTX 1080 在单机系统上进行集群, 目的是快速进行小规模数据的深度学习网络训练。

## 2. NVIDIA 显卡

到目前为止, NVIDIA 推出过的 GeForce 系列显卡多达数百张, 虽然其中大部分显卡已经被淘汰掉了, GeForce 系列现在流行的是 GTX 750、980、1060、1080。如何选择合适的显卡并使用多少张显卡来并行计算也是一个值得思考的问题。

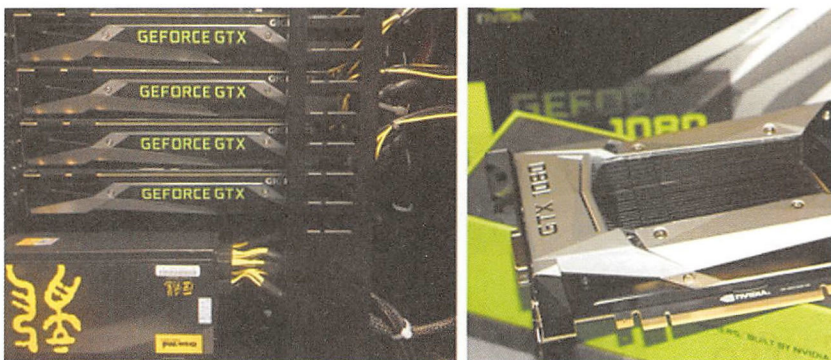


图 1-12 某公司的深度学习硬件平台：4 个 GTX 1080 集群

不同型号的 NVIDIA 显卡有各自对应的擅长之处。当需要处理的数据量比较小或者神经网络模型比较小时，可以选择频率高的 GPU；当需要处理数据量较大或者神经网络模型比较大时，可以选择显存大、处理器核心数较多的 GPU。如果有更高的精度要求，最好选择 Kepler、Volta、Pascal 架构的 GPU。在挑选时要注意几个参数：处理器核心、工作频率、显存位宽、单卡或双卡。有读者觉得显存位宽最重要，也有读者觉得核心数量最重要，作者认为对深度学习计算而言，处理器核心数和显存大小比较重要。这些参数越高越好，但最终的性能也与程序相关。如果程序无法让所有的寄存器和处理器核心都工作，资源就会被浪费。

GPU 架构有 Tesla、Fermi、Kepler、Maxwell、Pascal 和 Volta。GPU 架构指的是硬件的设计方式，例如流处理器簇中有多少个处理器核心、是否有 L1 或 L2 高速缓存、是否有双精度计算单元等。每一代的架构是一种思想，本质上都是去更好地完成并行的思想。NVIDIA 显卡系列有 GeForce、Quadro、Tesla。显卡系列在本质上并没有什么区别，只是 NVIDIA 区分成 3 种选择，GeForce 用于家庭娱乐，Quadro 用于工作站，而 Tesla 系列用于服务器。需要注意的是，Tesla 没有显示输出接口，它专注于数据计算而不是图形显示，因此不能用于电脑游戏。

## 1.4 深度学习的软件框架

通过学习上一节，我们知道了 GPU 为什么更加适合深度学习。选择好硬件加速器之后，剩下的就是软件问题了，即应该使用哪种深度学习框架。

深度学习领域的五大巨头都各自力挺一种深度学习框架：Google 有自家的



TensorFlow, Facebook 有 Torch, 百度有 PaddlePaddle, 微软有 CNTK, 而 Amazon 的 AWS 则有 MXNet, 现在还有支持 HADOOP 的 NL4J 深度学习框架 (见图 1-13)。



图 1-13 各大深度学习软件框架

表 1-1 是五大主流深度学习框架概要对比表。每种框架都有其优缺点, 选择的时候需要根据自身业务的实际需求。例如需要用到对时间序列分析的, 就使用循环神经网络 RNN, 而 Caffe 和 MXNet 对图像卷积处理非常友好, 但缺乏对循环神经网络的支持, Google 的 TensorFlow 则是支持其他机器学习算法和增强学习 (Reinforcement Learning) 算法。常言道“欲先攻必先利器”, 在正式进入深度学习之前, 让我们一起来了解一下各大主流深度学习框架之间的关系及其优缺点。当然, 在作者写书的过程中, 上述框架已经迭代更新了若干个版本, 部分论述可能已经过时, 本节知识仅供参考。

表 1-1 五大主流深度学习框架概要对比表

	开发语言	灵活性	文档	适合模型	难易度
Caffe 1.0	C++/CUDA	一般	全面	CNN	中等
TensorFlow	C++/CUDA/Python	好	中等	CNN/RNN/RL	困难
Torch	Lua/C/CUDA	好	全面	CNN/RNN	中等
Theano	Python/C++/CUDA	好	全面	CNN/RNN	容易
MXNet	C++/CUDA	好	全面	CNN	中等

## 1. Caffe 1.0

Caffe1.0 是第一个主流的工业级深度学习工具。在 2013 年年底, 由 UC Berkely 的贾扬清基于 C 和 C++ 开发的深度学习框架, 其特点是具有非常出色的卷积神经网络实现功能, 尤其在 2013 ~ 2016 年, 大部分与视觉有关的深度学习论文都采用了 Caffe 框架。至今为止, Caffe 在计算机视觉领域依然是最流行的工具包。可是因为开发较早和历史遗留问题, 其架构的缺点是不够灵活, 缺乏对递归网络 RNN 和语言建模的支



持，因此 Caffe 不适用于文本、声音或时间序列数据等类型的深度学习应用。

## 2. TensorFlow 0.88

TensorFlow 基于 Python 语言编写，通过 C/C++ 引擎加速，是 Google 开源的第二代深度学习框架。TensorFlow 处理递归神经网络 RNN 非常友好，并且内部实现使用了向量运算的符号图方法，使用图来表示计算任务，使得新网络的指定变得容易，支持快速开发。TensorFlow 的用途不止于深度学习，还可以支持增强学习和其他机器学习算法，扩展性很好。缺点是目目前 TensorFlow 还不支持“内联 (inline)”矩阵运算，必须要复制矩阵才能对其进行运算。复制庞大的矩阵会导致系统运行效率降低，并占用部分内存。另外，TensorFlow 不提供商业支持，仅为研究者提供的一种新工具，因此公司如果要商业化需要考虑开源协议的问题。

## 3. Torch

Torch 是由 Facebook 用 Lua 语言编写的开源计算框架，支持机器学习算法。其具有较好的灵活性和速度，实现并且优化了基本的计算单元，可以很简单地在此基础上实现自己的算法，不用在计算优化上浪费精力。Facebook 于 2017 年 1 月开放了 Torch 的 Python API——PyTorch 源代码，其支持动态计算图，能处理长度可变的输入和输出，尤其适用于循环神经网络 RNN。缺点是底层为 Lua 语言，如果需深入了解其内部工作方式需要时间去学习新的编程语言。

## 4. Theano

Theano 是深度学习框架中的元老，使用 Python 编写。Theano 派生出了大量 Python 深度学习库，最著名的包括 Blocks 和 Keras。其最大特点是非常灵活，适合做学术研究的实验，且对递归网络和语言建模有较好的支持，缺点是速度较慢。

## 5. MXNet

MXNet 主要由 C/C++ 语言编写，提供多种 API 的机器学习框架，面向 R、Python 和 Julia 等语言，目前已被 Amazon 云服务作为其深度学习的底层框架。由于 MXNet 是 2016 年新兴的深度学习框架，因此大量借鉴和避免了 Caffe 的优点和缺点。其最主要的特点是具有分布式机器学习通用工具包 DMLC，因此其分布式能力较强。MXNet 还注重灵活性和效率，文档也十分详细，同时强调提高内存的使用效率，甚至能在智能手机上运行诸如图像识别等任务。但是其与 Caffe 一样缺乏对循环神经网络 RNN 的支持。在分布式方面，没有使用 JAVA 实现的 NL4J 方便。

## 6. Keras

Keras 是一个基于 Theano 和 TensorFlow 的深度学习库。由于受到深度学习元老框架 Torch 的启发，Keras 拥有较为直观的 API，有望成为未来开发神经网络的标准

Python API。本书主要采用 Keras 作为主要代码 API，因为其具有简洁的 API 接口，方便理解深度学习的原理与插入代码片段。

表 1-2 所示是五大主流深度学习框架优缺点对比表。

表 1-2 五大主流深度学习框架优缺点对比表

	优点	缺点
Caffe 1.0	<ul style="list-style-type: none"> <li>(1) 适合前馈网络和图像处理；</li> <li>(2) 适合微调已有的网络；</li> <li>(3) 定型模型，无须编写任何代码</li> </ul>	<ul style="list-style-type: none"> <li>(1) 不适合循环网络；</li> <li>(2) 用于大型 CNN 网络，操作过于烦琐；</li> <li>(3) 扩展性差，不够精简；</li> <li>(4) 更新缓慢</li> </ul>
TensorFlow	<ul style="list-style-type: none"> <li>(1) 计算图抽象化，易于理解；</li> <li>(2) 编译时间快于 Theano；</li> <li>(3) 用 TensorBoard 进行可视化；</li> <li>(4) 支持数据并行和模型并行</li> </ul>	<ul style="list-style-type: none"> <li>(1) 速度较慢，内存占用较大；</li> <li>(2) 不提供商业支持；</li> <li>(3) 已预定型的模型不多；</li> <li>(4) 不易于工具化；</li> <li>(5) 在大型软件项目中容易出错</li> </ul>
Torch	<ul style="list-style-type: none"> <li>(1) 大量模块化组件，易于组合；</li> <li>(2) 易于编写自定义层；</li> <li>(3) 预定型的模型很多</li> </ul>	<ul style="list-style-type: none"> <li>(1) 要学习 Lua 和使用 Lua 作为主语言；</li> <li>(2) 即插即用，代码相对较少；</li> <li>(3) 不提供商业支持；</li> <li>(4) 文档质量不高</li> </ul>
Theano	<ul style="list-style-type: none"> <li>(1) Python + NumPy 实现，接口简单；</li> <li>(2) 计算图抽象化，易于理解；</li> <li>(3) RNN 与计算图配合好；</li> <li>(4) 很多高级包派生，例如 Keras</li> </ul>	<ul style="list-style-type: none"> <li>(1) 原始的 Theano 级别偏低；</li> <li>(2) 大型模型的编译时间可能较长；</li> <li>(3) 对已预定型模型的支持不够完善；</li> <li>(4) 只支持单个 GPU</li> </ul>
MXNet	<ul style="list-style-type: none"> <li>(1) 适合前馈网络和图像处理；</li> <li>(2) 适合微调已有的网络；</li> <li>(3) 定型模型，无需编写任何代码；</li> <li>(4) 有更多学界用户对接模型；</li> <li>(5) 支持 GPU、CPU 分布式计算</li> </ul>	<ul style="list-style-type: none"> <li>(1) 不适合循环网络 RNN；</li> <li>(2) 群集运维成本比 DL4J 高</li> </ul>

根据作者的实际开发经验，未来的深度学习模型可以应用于大型的服务器和分布式集群。建议采用 MXNet 或者 DL4J，如果需要与 Hadoop 的业务相结合，DL4J 或许是最好的选择。如果只懂 C 或者 C++ 语言，不懂 Python 语言，Caffe 或许是最好的选择。如果是非商业开发，可以选择 Google 的 TensorFlow 或者 Facebook 的 Torch，具体需要根据不同的语言进行选择。对于教学和 Demo 实验，可以使用 Keras 去实现简单的深度学习模型。最后遗憾的是在 2017 年下半年，Yoshua Bengio 宣布将

停止更新维护 Theano。

## 1.5 本章小结

人工智能发展了 50 多年，衍生了至今仍非常火热的机器学习，机器学习中的神经网络又衍生出了让学习更深、更广、更精准的深度学习。正是因为深度学习可以自动挖掘数据中的深层次高维度信息，降低了工程师们大量的劳动时间，并进一步提高预测的准确率。作为新一代算法工程师的我们更加迫切需要学习深度学习，从而“解放”自己。

在真正进入深度学习之前，很有必要了解一下硬件对于深度学习的作用。通过简述为什么 GPU 比 CPU 更加适合深度学习，让我们知道了 GPU 加速器对于深度学习来说是必不可少的。了解完深度学习的硬件选择之后，比较不同的深度学习的软件框架。对于一个以实际目标场景为驱动的工程师来说，不需要重新发明深度学习软件框架，需要的是对比现有的深度学习软件框架，然后选择一款最合适的进行了解和开发，这才是上上之策。

- 监督学习：一般方法分为两个阶段——训练和预测。通过使用算法对结构化或者非结构化的数据进行解析，从数据中学习训练，获取数据中特定的结构模型，然后使用这些结构或者模型来对未知的新数据进行预测。
- 深度学习：具有庞大的神经元、复杂的网络连接方式、惊人的计算量、自动提取特征。
- 硬件加速器：主要分为 GPU、ASIC 和 FPGA 这 3 种。
- GPU 特性：高带宽、高速的缓存性能、并行单元多。

## 引用/参考

- [1] Bengio Y, Courville A, Vincent P. Representation Learning: A Review and New Perspectives[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2013, 35(8):1798-828.
- [2] Schmidhuber J. Deep learning in neural networks: An overview[J]. Neural Netw, 2014, 61:85-117.
- [3] Lecun Y, Bengio Y, Hinton G. Deep learning[J]. Nature, 2015, 521(7553):436.
- [4] Schmidhuber J. Deep Learning[J]. Scholarpedia, 2016, 10(11).
- [5] Olshausen B A, Field D J. Emergence of simple-cell receptive field properties by learning a sparse code for natural images.[J]. Nature, 1996, 381(6583):607-609.
- [6] Schmidhuber J. Multi-column deep neural networks for image classification[C]// Computer

Vision and Pattern Recognition. IEEE, 2012:3642-3649.

- [7] Ivakhnenko A G, Lapa V G. CYBERNETIC PREDICTING DEVICES,[J]. Transdex, 1966.
- [8] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C]// International Conference on Neural Information Processing Systems. Curran Associates Inc. 2012:1097-1105.
- [9] Deng L, Yu D. Deep Learning: Methods and Applications[J]. Foundations & Trends in Signal Processing, 2014, 7(3):197-387.
- [10] He K, Gkioxari G, Dollár P, et al. Mask R-CNN[J]. 2017.



# 第 2 章

## 人工神经网络

本章主要内容：

- 人工神经网络介绍
- 模型的训练与预测
- 人工神经网络的核心算法

本章我们将要深入机器学习领域当中很重要的分支方向——人工神经网络(Artificial Neural Network, ANN)，它是现今最火热的人工智能研究方向——深度学习的基础。

现今识别准确率最高的手写字体识别系统是用什么算法模型实现的？在工业控制系统当中，如何求解约束优化问题？机场的作业调度系统那么复杂，用什么机器学习算法去控制？对于不确定的实时控制系统，如何反馈信息才能让控制系统达到动态平衡？上面的问题都是人工神经网络的实际应用场景，它既可以处理小规模数据模型，也能处理复杂的控制系统。

人工神经网络以其独特的网络结构和处理信息的方法，在自动控制领域、组合优化问题、模式识别、图像处理、自然语言处理等诸多领域，已经取得了辉煌的成绩。

在本章中，我们从人工神经模拟生物神经元开始切入，再到人工神经元组合构成人工神经网络模型。仅仅了解人工神经网络的基本模型是不够的，我们需要进一步深入学习人工神经网络的核心算法，掌握模型的训练与预测。最后利用本章的知识设计并实现一个人工神经网络对医疗数据进行分类。

如果你只是了解机器学习中的聚类、回归、分类等常用的算法，但觉得人工神经网络神秘莫测，那么紧跟我们的脚步，一起深入探索人工神经网络的奥妙。如果你已经掌握了人工神经网络的基本知识，那就再好不过了，因为人工神经网络是深度神经网络中最经典，也是最基础的网络模型。接下来让我们一起开启深度学习的基础旅程，感受深度学习的魅力！

## 2.1 人工神经网络概述

### 2.1.1 历史背景

神经网络已经发展了近 70 年，其技术发展路线颇为波折，经历了 3 次高潮，分别是 1940 ~ 1960 年、1980 ~ 1990 年、2006 年至今。

早在 20 世纪 40 年代初，著名的控制论学家 Warren Mcculloch 和逻辑学家 Walter Pitts 在分析并总结生物神经元基本特性后，通过阈值逻辑的数学算法设计出了神经网络模型。此后，神经网络的研究分为对大脑生物过程的研究和把神经网络应用于人工智能的研究两条不同的路线。该神经网络模型沿用至今，并直接影响着相关领域研究的进展。

20 世纪 40 年代末，心理学家 Donald Hebbian 根据生物神经的可塑性机制提出了

Hebbian 学习,被认为是一种典型的无监督学习规则,后来其他研究人员把这种计算模型的思想应用到了图灵 B 型机上。到了 1954 年, Farley 和 Clark 首次使用计算机模拟了 Hebbian 网络,同期有着众多科学家,如 Rochester、Holland、Habit 等发明了大量的神经网络计算器,神经网络模型从萌芽期进入了第一个高潮时期。

1969 年,神经网络的发展进入了停滞期,因为 Minsky 和 Papert 发现了神经网络计算模型的两个重大缺陷:一是计算资源严重制约神经网络,无法计算大型神经网络;二是基本感知机无法处理异或回路。以上两个缺陷促使大批的研究人员对神经网络的发展前景失去了信心,神经网络的研究迈入第一次低谷期。

1975 年, Werbos 发表反向传播算法,该算法能够有效地解决感知机异或回路的问题,使得训练多层的神经网络模型变成了现实。从反向传播算法的发表开始,神经网络迎来第二个高潮。20 世纪 80 年代中期,分布式并行处理算法逐渐流行起来, Rumelhart 和 McClelland 描述了如何使用连接器去模拟神经元的处理过程。1989 年, Yann Lecun 等研究人员实现了一个 7 层的卷积神经网络 LeNet-5,识别手写数字的精度达到了 98%。在这段时间内,一大批学者和研究人员围绕 Hopfield 提出的人工神经网络方法展开了进一步研究,形成了 20 世纪 80 年代中期以来人工神经网络的研究热潮。

可是好景不长,在 20 世纪末,支持向量机 SVM 和其他机器学习算法的流行程度逐渐高涨,更重要的是其他机器学习算法在实现效率上远超神经网络。随着 PageRank 等互联网新兴的算法不断被提出,学者们的研究重心也开始转移到其他机器学习算法上,神经网络的研究热潮又迎来了第二个低谷。

2006 年, Hinton 等人利用限制玻尔兹曼机对神经网络的连续层进行建模,使用逐层预训练的方法抽取模型数据中的高维特征,后来又提出了深度信念网络。这一技巧随后被众多学者推广到了许多不同的神经网络架构上,大大地提高了模型在测试集上的泛化效果。同时,随着硬件计算能力和算法的提升,网络隐层可以不断增加,于是以 Hinton 为代表的研究人员重新将人工神经网络定义为深度学习,并进行推广。

2009 ~ 2012 年, Jurgen Schmidhuber 在 Swiss AI Lab IDSIA 的研究小组发表的递归神经网络和深前馈神经网络赢得了 8 项关于模式识别和机器学习的国际比赛的奖项。同期, Alex Graves et al. 的双向、多维 LSTM 网络模型赢得了 2009 年 ICDAR 的 3 项关于连笔字识别的比赛的奖项。2012 年, Ng 和 Dean 创建了一个独特的神经网络模型,学会了识别更高层次的抽象数据,能够对猫、狗进行分类,并从 YouTube 视频流中识别出视频的内容。

2006 年至今,神经网络迎来了第三次高潮并重新被命名为深度学习。这一次神经网络的高潮与开源运动相结合,衍生了一系列的深度学习开源框架,才有了今天



深度学习在无人驾驶、机器翻译、金融风控等诸多领域的广泛应用。

## 2.1.2 基本单位——神经元

提到神经网络的历史，不得不提起神经网络最开始的设计思路。受生物学的启发，人工神经网络是生物神经网络的一种模拟和近似。它从结构、实现机理和功能上模拟生物神经网络，传统的生物神经元模型由树突、细胞核、细胞体、突触和神经末梢组成，如图 2-1 所示。

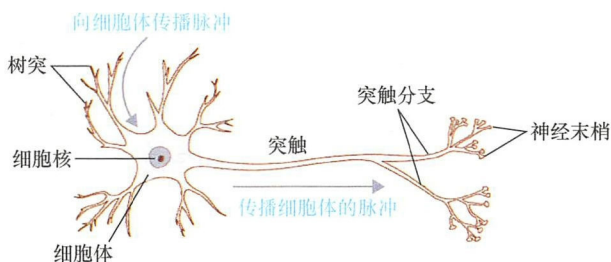


图 2-1 生物神经元模型示例图

那么人工神经元是如何模仿生物神经元的呢？它与生物神经元模型之间的区别是什么？

如图 2-2 所示，神经元的输入  $x_i$  对应生物神经元的树突。输入  $x_i$  向细胞体传播脉冲，相当于输入权值参数  $w_i$ ，通过细胞核对输入的数据和权值参数进行加权求和。传播细胞体的脉冲相当于人工神经元的激活函数，最终输出结果  $y$  作为下一个神经元的输入。

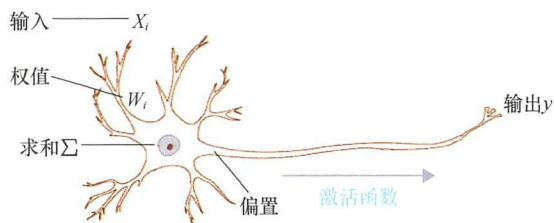


图 2-2 模拟生物神经元模型的人工神经元模型示例图

上述即为人工神经网络中最基本的处理单元——神经元，主要由连接、求和节点、激活函数组成。

- 连接（Connection）：神经元中数据流动的表达方式。

- 求和结点 (Summation Node): 对输入信号和权值的乘积进行求和。
- 激活函数 (Activate Function): 一个非线性函数, 对输出信号进行控制 (后文将会详细介绍激活函数)。

为了方便理解, 我们对生物神经元抽象, 得到了神经元基本模型 (见图 2-3), 其中:

- $x_1, x_2, \dots, x_n$  为输入信号的各个分量;
- $w_1, w_2, \dots, w_n$  为神经元各个突触的权值;
- $b$  为神经元的偏置参数;
- $\Sigma$  为求和节点,  $z = x_1 w_1 + x_2 w_2 + \dots + x_n w_n + b = \sum_{i=1}^n w_i x_i + b$ ;
- $f$  为激活函数, 一般为非线性函数, 如 Sigmoid、Tanh 函数等;
- $y$  为该神经元的输出。

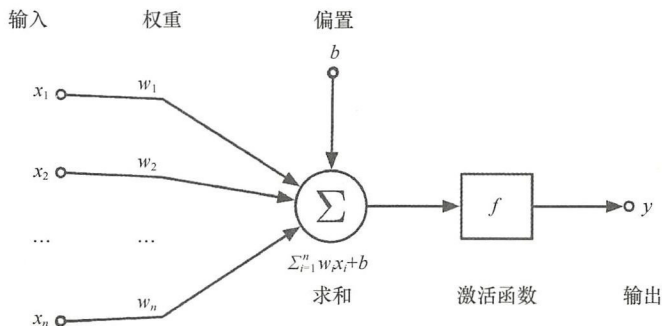


图 2-3 神经元的基本模型

上述神经元的基本模型有些复杂, 我们使用数学表达式对神经元的基本模型继续进行抽象, 得到神经元的数学基本表达式为:

$$y = f\left(\sum_{i=1}^n w_i x_i + b\right) \quad (2-1)$$

假设  $\mathbf{W} = [w_1, w_2, \dots, w_n]$  为权值向量,  $\mathbf{X} = [x_1, x_2, \dots, x_n]$  为输入向量。一个神经元的基本功能是对输入向量  $\mathbf{X}$  与权值向量  $\mathbf{W}$  内积求和后并加上偏置参数  $b$ , 经过非线性的激活函数  $f$ , 得到  $y$  作为输出结果。因此神经元又可以使用矩阵形式表达为:

$$y = f(\mathbf{W}^T \mathbf{X} + b) \quad (2-2)$$

### 2.1.3 线性模型与激活函数

无论是线性回归模型, 还是分类模型, 本质上都是把数据进行映射, 既可以映射到一

个或者多个离散的标签上，也可以映射到连续的空间里。单个神经元模型在 没有加入激活函数之前，可以看作一个线性回归模型，把输入的数据映射到一个  $n$  维的平面空间中。

在线性模型中，模型的输出为输入的加权和。假设一个模型的输出  $y$  和输入属性  $x_i$  满足式 (2-3)，那么该模型就是一个线性模型。

$$y = \sum_i w_i x_i + b \quad (2-3)$$

其中， $w_i$  和  $b$  为线性模型的参数。因此，线性模型具有很好的解析性，参数  $w_i$  表示每个属性  $x_i$  在回归过程中的重要程度。

线性模型的特点是任意线性模型的组合仍然是线性模型。当  $i=1$  时，式 (2-3) 为  $y = wx + b$ ，形成平面坐标系上的一条直线地；类似地，当  $i=n$  时，模型中的输入  $x$  和输出  $y$  形成在  $n+1$  维空间中的一个平面。细心的读者会发现该线性模型实际上就是 2.1.2 节神经元模型中的一部分（神经元没有加入激活函数之前）。

可遗憾的是，对于现实世界的数据来说，很多时候数据都是线性不可分的，因此需要非线性模型。一个没有激活函数的神经元模型只是一个线性回归模型，它的表达能力有限，不能表示复杂的数据分布。因此，我们需要在神经元中加入一个非线性函数（称为激活函数），这就相当于为该神经元引入了非线性因素，从而使神经元模型更好地解决复杂的数据分布问题（见图 2-4）。

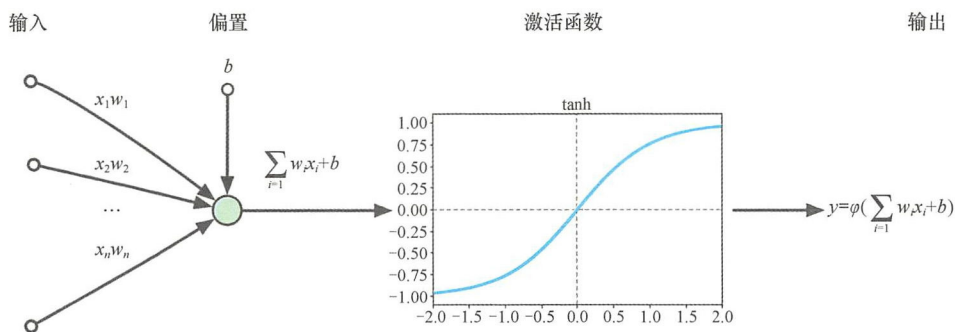


图 2-4 神经元模型。在没有加入激活函数时，神经元模型实际上是一个线性回归模型。加入激活函数后，该神经元模型可以解决复杂的数据问题

神经元模型不仅可以看作对生物神经元的模拟，而且能够有效处理数学中的非线性数据。理论上，由多个神经元组合而成的神经网络可以表示成任何复杂的函数。

## 2.1.4 多层神经网络

人工神经网络由许多的神经元组合而成，神经元组成的信息处理网络具有并行



分布结构，因此有了更复杂的人工神经网络。

一个神经网络由多个神经元结构组成，每一层的神经元都拥有输入和输出，每一层都是由多个神经元组成。第  $l-1$  层网络神经元的输出是第  $l$  层神经元的输入，输入的数据通过神经元上的激活函数来控制输出数值的大小。该输出数值是一个非线性值，通过激活函数求得数值，根据极限值来判断是否需要激活该神经元。

一般多层人工神经网络 ANN 由输入层、输出层和隐藏层组成。

- 输入层 (Input Layer): 接收输入信号作为输入层的输入。
- 输出层 (Output Layer): 信号在神经网络中经过神经元的传输、内积、激活后，形成输出信号进行输出。
- 隐藏层 (Hidden Layer): 隐藏层也被称为隐层，它介于输入层和输出层之间，是由大量神经元并列组成的网络层，通常一个神经网络可以有多个隐层。

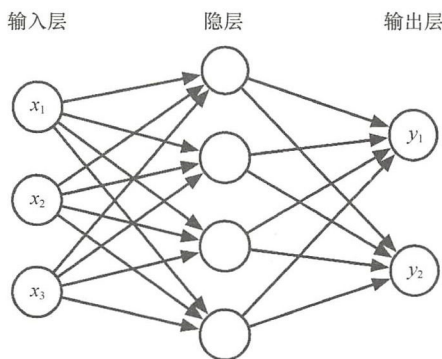


图 2-5 3 层人工神经网络模型

图 2-5 所示为一个 3 层结构的人工神经网络模型。从左到右开始，第 1 层为输入层，输入向量为  $[x_1, x_2, x_3]$ ；第 2 层为带有 4 个节点的隐层；第 3 层为输出层，输出向量为  $[y_1, y_2]$ 。

其中，输入层为 3 个元素组成的一维向量，隐层有 4 个节点，因此从输入层到隐层共有  $3 \times 4 = 12$  条连接线；输出层是由 2 个元素组成的一维向量，因此从隐层到输出层共有  $4 \times 2 = 8$  条连接线。

在进一步了解人工神经网络之前，我们需要知道如下内容。

- 神经网络输入层与输出层的节点数往往是固定的，根据数据本身而设定，隐层数和隐层节点则可以自由指定。
- 神经网络模型图中的箭头代表预测过程时数据的流向，与学习训练阶段的数据流动方向不同。
- 神经网络的关键不是节点而是连接，每层神经元与下一层的多个神经元相连接，每条连接线都有独自的权重参数。此外，连接线上的权重参数是通过训练得到的。
- 神经网络中不同层的节点间使用全连接的方式，也就是  $l-1$  层的所有节点对于  $l$  层的所有节点都会有相互连接。例如当前层有 3 个节点，下一层 4 个节点，那么连接线有  $3 \times 4 = 12$  条，共 12 个权重值和 4 个偏置。

## 2.2 训练与预测

在第1章的图1-4中，深度学习的一般方法分为训练与预测阶段。在训练阶段，我们需要准备好原始数据和与之对应的分类标签数据，通过训练得到模型A。在预测阶段，对新的数据套用该模型A，可以预测新输入数据所属类别。在人工神经网络中，该模型A的内部结构使用了类似图2-5的多层神经网络。

### 2.2.1 训练

在设计人工神经网络模型结构前，我们需要对输入和输出的数据进行量化。假设输入的数据有 $k$ 个，输出为 $n$ 个分类，那么输入层的神经元节点数应设置为 $k$ ，输出层的神经元节点数应设置为 $n$ 。隐层神经元节点的数目不确定，但其数目越多，神经网络的非线性越显著，从而增强人工神经网络的健壮性。习惯上，一般第 $l$ 层神经元的节点数为 $l-1$ 层节点数的 $1 \sim 1.5$ 倍。

人工神经网络模型结构定义好后，也就确定了输入层、输出层、隐层的节点数。剩下没有被确定的参数有权值向量 $W$ 和偏置 $b$ ，那么神经网络模型中的权值向量 $W$ 和偏置 $b$ 是如何确定的呢？

这就是训练过阶段的目标——确定权值向量 $W$ 和偏置 $b$ （训练过程也被称为神经网络的学习过程）。

人工神经网络的训练实际上是通过算法不断修改权值向量 $W$ 和偏置 $b$ ，使其尽可能与真实的模型逼近，以使得整个神经网络的预测效果最佳。

具体做法：首先给所有权值向量 $W$ 和偏置 $b$ 赋予随机值，使用这些随机生成的权重参数值来预测训练数据中的样本。样本的预测值为 $\hat{y}$ ，真实值为 $y$ 。现在定义一个损失函数，目标是使预测值 $\hat{y}$ 尽可能接近于真实值 $y$ ，损失函数就是使神经网络的损失值和尽可能小。其基本公式为：

$$\text{loss} = (\hat{y} - y)^2 \quad (2-4)$$

现在问题转变为：如何改变神经网络中的参数 $W$ 和 $b$ ，让损失函数的值最小？

如何求损失函数的最小值，最终演化成为一个优化问题。对神经网络的优化就是对参数的优化，减少损失，直至损失收敛。为了解决该优化问题，研究者发现可以使用梯度下降算法来优化网络中的参数 $W$ 和 $b$ 。可是问题又来了，人工神经网络的模型结构复杂，计算网络中的所有参数梯度不是一件容易的事情。于是又引入了反向传播算法，最终可以使用反向传播算法求得网络模型所有参数的梯度，通过梯度下降算法对网络的参数进行更新。

当损失函数收敛到一定程度时结束训练，保存训练后神经网络的参数。

## 2.2.2 预测

在训练阶段，通过算法修改神经网络的权值向量  $W$  和偏置  $b$ ，使得损失函数最小。当损失函数收敛到某个阈值或者等于零时，停止训练，就可以得到确定的权值向量  $W$  和偏置  $b$ 。

到此为止，人工神经网络结构中所有的参数（输入层、输出层、隐层的节点数，权值向量  $W$ ，偏置  $b$ ）都是已知的，只需要将向量化后的数据从人工神经网络的输入层开始输入，顺着数据流动的方向在网络中进行计算，直到数据传输到输出层并输出（一次向前传播），就完成一次预测并得到分类结果。

## 2.3 核心算法

神经网络在训练和预测阶段都需要频繁使用向前传播算法，而在训练阶段则多次提到了梯度下降算法和反向传播算法。下面将对人工神经网络的三大核心算法（梯度下降算法、向前传播算法、反向传播算法）进行详细解释。

### 2.3.1 梯度下降算法

在微积分中，对多元函数的参数求偏导数，求得参数的偏导数以向量的形式表达就是梯度。如图 2-6 所示， $\theta_0$  到  $\theta_1$  的距离为  $\theta_0$  在函数  $L(\theta)$  上的梯度，记作  $\partial L / \partial \theta$ 。

那这个梯度向量  $\partial L / \partial \theta$  有什么意义呢？数学上，梯度越大，则函数的变化越大。对于函数  $L(\theta)$  在点  $\theta_0$  处，梯度向量  $\partial L / \partial \theta$  的方向就是函数  $L(\theta)$  增加最快的方向。也就是说，沿着梯度向量的方向易于找到函数的最大值。反之亦然，沿着与梯度向量相反的方向，梯度减少最快，易于找到函数的最小值。

假设函数  $L(\theta)$  为损失函数，为了找到损失函数的最小值，需要沿着与梯度向量

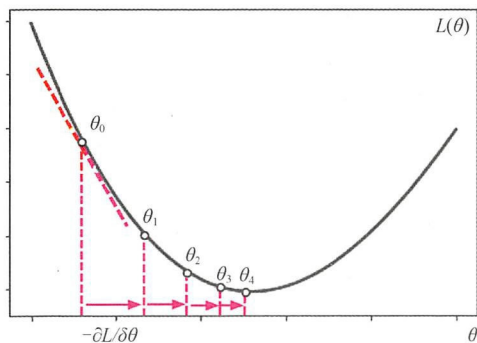


图 2-6 梯度下降算法



相反的方向  $-\partial L / \partial \theta$  更新变量  $\theta$ ，这样可以使得梯度减少最快，直至损失收敛至最小值。该算法称为梯度下降算法，其基本公式为：

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta} \quad (2-5)$$

其中， $\eta \in \mathbf{R}$  为学习率，用于控制梯度下降的幅度（快慢）。在神经网络中，式 (2-5) 中的变量  $\theta$ ，为由神经网络的参数所组成的向量  $\theta = \{w_1, w_2, \dots, w_n, b_1, b_2, \dots, b_n\}$ 。

梯度下降的目标就是找到参数  $\theta^*$  使得神经网络总损失  $L$  最小，从而确定神经网络中的权重向量  $W$  和偏置  $b$ 。

继续回到图 2-6，梯度下降算法每次计算参数  $\theta_i$  在当前位置的梯度，然后让参数  $\theta_i$  顺着梯度的反方向前进一段距离，不断重复该过程。直到梯度接近于零的时候，算法认为找到了损失函数  $L(\theta)$  的最小值并停止计算。此时可以认为参数  $\theta^*$  恰好到达让损失函数位于最小值的状态，也就是神经网络的预测值接近于真实值的状态。

梯度下降算法的特点是：越接近目标值，变化越小，梯度下降速度越慢。实际上，梯度下降算法的变种有很多，下面对常用的梯度下降算法进行介绍。

### 1. 批量梯度下降算法 (Batch Gradient Descent, BGD)

批量梯度下降算法是梯度下降算法中最常用的形式，所有样本都参与参数  $\theta$  的更新。假设有  $m$  个样本， $m$  个样本都参与调整参数  $\theta$ ，因此得到的是一个标准的梯度。

- 优点：易于得到全局最优解；总体迭代次数不多。
- 缺点：当样本数目很多时，训练时间过长，收敛速度变慢。

### 2. 随机梯度下降算法 (Stochastic Gradient Descent, SGD)

随机梯度下降算法原理与批量梯度下降算法类似，区别在于随机梯度是从  $m$  个样本中随机抽取  $n$  个样本求解其梯度。

- 优点：训练速度快，每次迭代计算量少。
- 缺点：准确度下降，得到的不一定是全局最优；总体迭代次数比较多。

### 3. 小批量随机梯度下降算法 (Min-batch SGD)

批量梯度下降算法与随机梯度下降算法在样本选取上不同，优缺点明显。

- 在训练速度上，随机梯度算法每次采用一个样本进行迭代，因此训练速度快；而批量梯度下降算法则因为样本量大，训练速度不能让人满意。
- 在收敛速度上，随机梯度下降算法一次迭代的样本量有限，导致梯度变化很大，不能快速收敛到局部最优解。
- 在准确率上，随机梯度下降算法使用部分样本来决定梯度方向，导致得到的可能不是全局最优解，而是局部最优解。

小批量随机梯度下降算法是对批量梯度下降算法和随机梯度下降算法的折衷方法：每次随机从  $m$  个样本中抽取  $k$  个进行迭代求梯度，每一次迭代的抽取方式都是

随机的，因此部分样本会重复。这样做的好处是计算梯度时让数据与数据之间产生关联，避免数据最终只能收敛到局部最优解。

如图 2-7 所示，批量梯度下降算法的下降轨迹平滑，经过多次迭代后达到全局最优解处；而小批量随机梯度下降算法虽然在下降过程有持续的小幅振荡，但每次迭代的数据量少，能够更快地找到全局最优解。因此小批量随机梯度下降算法在实际工程当中使用的频率更高。

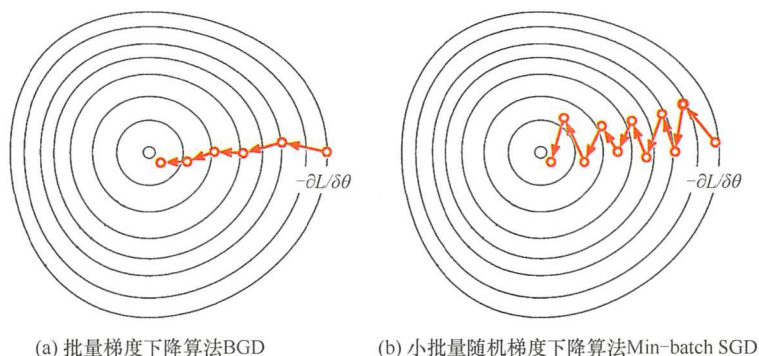


图 2-7 BGD 和 Min-batch SGD 梯度下降示例图

## 2.3.2 向前传播算法

向前传播（Feed Forward）算法在神经网络的训练和预测阶段会被反复使用，是神经网络中最常见的算法。其计算方式简单，只需要根据神经网络模型的数据流动方向对输入的数据进行计算，最终得到输出的结果。

图 2-8 所示为一个简单的人工神经网络模型，其输入层有 2 个节点，隐层、输出层均有 3 个节点。下面我们通过矩阵的方式对向前传播算法进行解析。

假设  $z_i^{(l)}$  为第  $l$  层网络第  $i$  个神经元的输入， $w_{i,j}^{(l)}$  为第  $l$  层网络第  $i$  个神经元到第  $l+1$  层网络中第  $j$  个神经元的连接，那么根据神经元基本公式有：

$$z_i^{(l)} = \sum_{j=1} w_{i,j}^{(l)} x_j + b_i \quad (2-6)$$

对图 2-8 的神经网络的参数进行向量化，表示为：

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix}, X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

那么有：

$$\begin{aligned}
 W^T X + B &= \begin{bmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \\ w_{1,3} & w_{2,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \\
 &= \begin{bmatrix} w_{1,1}x_1 + w_{2,1}x_2 \\ w_{1,2}x_1 + w_{2,2}x_2 \\ w_{1,3}x_1 + w_{2,3}x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \\
 &= \begin{bmatrix} \sum_{i=1}^2 w_{i,1}x_i + b_1 \\ \sum_{i=2}^2 w_{i,1}x_i + b_2 \\ \sum_{i=3}^2 w_{i,1}x_i + b_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}
 \end{aligned}$$

因此，单层神经网络的向前传播算法可以使用矩阵表达为：

$$a^{(l)} = f(w^T x + b) \quad (2-7)$$

其中  $a^{(l)}$  为第  $l$  层网络的输出矩阵， $f$  为激活函数。

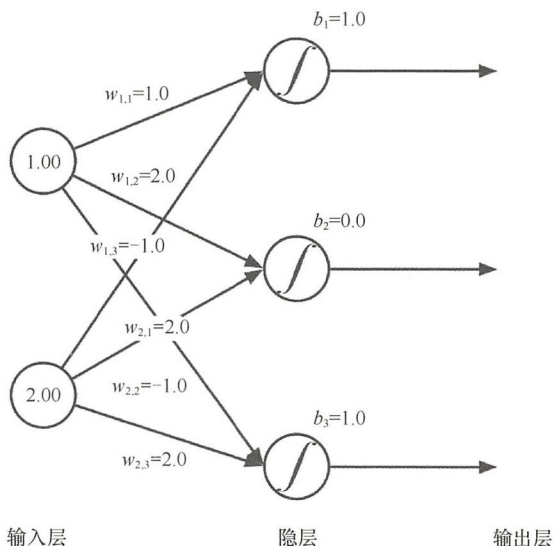


图 2-8 单隐层神经网络的向前传播。该神经网络的输入层节点数为 2，隐层节点数为 3

【代码清单 2-1】对应图 2-8，输入向量为 input\_a，输入层的权重向量为 weight，



偏置为  $b$ 。为了使代码简洁，激活函数 `sigmoid()` 使用了 Python 的 `lambda` 表达式。根据神经网络的矩阵计算公式  $y = f(w^T x + b)$ ，可以得到下一层网络节点的输出向量 `output_a`。

### 【代码清单 2-1】向前传播算法

```
def feedForward(input_a, weight, bias):
    f = lambda x: 1./1.+np.exp(-x) # 使用 Sigmoid 函数作为激活函数

    output_a = f(np.dot(weight, input_a) + bias)
    return output_a
```

## 2.3.3 反向传播算法

为什么会有反向传播（Back Propagation, BP）算法呢？如图 2-9 所示，假设神经网络中的一个参数  $w_{24}^l$  发生了小幅度的改变，那么这个改变将会影响后续激活值的输出，下一层根据这个激活值继续往后传递该改变。如此一层一层地往后传递，直到损失函数接收到该改变为止。最终这个小幅度的改变影响了整个网络，犹如在水面上划了一下，水波向外扩散影响了整个湖面的平静。

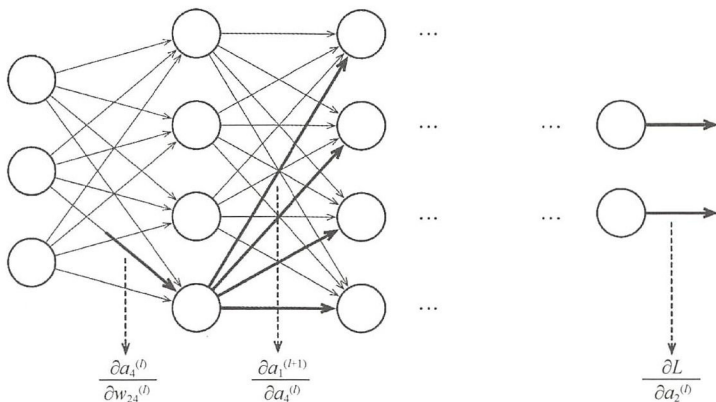


图 2-9 反向传播算法中使用的变量示例

为了知道这个改变的幅度，我们选择从网络的最后一层（输出层）开始，利用损失函数向前求得每一层每一个神经元的误差，从而获得这个参数的梯度（原始的小幅度的改变值）。

由于神经网络的结构模型复杂——多层网络进行排列堆叠，如何合理计算损失函数的梯度是一个严峻的挑战。梯度下降算法从整体出发，求得网络中参数的梯度

后对参数进行更新，而如何求得网络参数的梯度则是反向传播算法需要做的事情。

虽然在神经网络中，求网络参数的梯度不一定使用反向传播算法，并且反向传播算法有收敛慢、容易陷入局部最小解等缺点，但是其易用性、准确率却是其他求解梯度算法无法比拟的。因此在现代神经网络中，反向传播算法被广泛使用。

为了便于理解反向传播算法，这里举一个不太恰当的例子。现在有一个口述绘画游戏，第1个人看一幅画（输入数据），描述给第2个人（隐层）……依次类推，到最后一个人（输出）的时候，画出来的画肯定与原画不太像（误差较大）。下面开始反向传播算法，我们把画拿给最后一个人看（计算损失函数），然后最后一个人告诉前面一个人下次描述时需要注意哪些地方画错了（通过损失计算梯度）……同样依次类推，直到第1个人。当然该例子可能会与真正的反向传播算法有一定的差别，下面我们来看其数学定义。

假设神经网络模型的参数为  $\theta = \{w_1, w_2, \dots, w_n, b_1, b_2, \dots, b_n\}$ ，那么反向传播算法的精髓是：通过链式求导法则，求出网络模型中的每个参数的导数  $\partial L / \partial w_i$  和  $\partial L / \partial b_i$ 。

由于神经网络模型结构复杂，随着网络层数和节点数的增加，网络中的参数也会越来越多。为了便于计算机对神经网络中的参数进行跟踪和存储记录，我们可以通过计算图（Computation Graph）来对反向传播算法进行初步理解。

### 1. 理解计算图

计算图可以看作一种用来描述计算函数的语言，图中的节点代表函数的操作，边代表函数的输入。如图 2-10 所示，计算图的函数为：

$$f(x, y, z) = (x + y) \cdot z \quad (2-8)$$

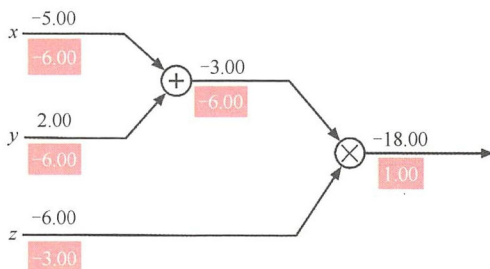


图 2-10 公式  $(x+y) \cdot z$  的计算图

反向传播算法希望通过链式法则得到式 (2-3) 中  $f$  的各个参数的梯度： $\partial f / \partial z$ 、 $\partial f / \partial x$ 、 $\partial f / \partial y$ 。

设  $q = x + y$ ，有： $\partial q / \partial x = 1$ ， $\partial q / \partial y = 1$ ；

同理，设  $f = qz$ ，有： $\partial f / \partial q = z$ ， $\partial f / \partial z = q$ 。

假设  $x$ 、 $y$ 、 $z$  输入的数据分别是  $-5$ 、 $2$ 、 $-6$ ，根据链式法则，我们可以得到式 (2-3) 中的各个参数的梯度值为：

$$\frac{\partial f}{\partial z} = q = -3$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = z \cdot 1 = -6$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = z \cdot 1 = -6$$

图 2-11 所示为单个神经元模型的计算图展开示例， $L$  为神经元的输出， $x_0$  和  $x_1$  分别为该神经元的输入， $z$  为求和节点。

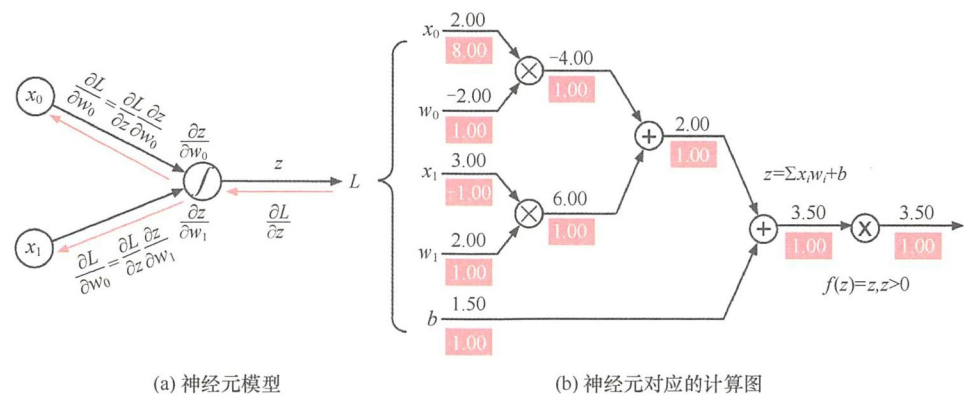


图 2-11 神经元展开计算图示例图。(a) 为带有两个输入的神经元模型，(b) 为该神经元模型展开的图计算公式  $y=1 \cdot (\sum_i w_i x_i + b)$ ，其中激活函数为  $f(z)=z, z>0$

从图 2-11 中可以看出，神经网络的反向传播算法就是利用计算图的原理，从网络的输出层开始，反向计算神经网络中每一个参数的梯度。然后通过梯度下降算法，以一定的学习率根据式 (2-5) 对神经网络中的参数进行更新。接着运行一次向前传播算法，得到新的损失值。对上述步骤不断迭代，就是整个神经网络训练的过程。

反向传播算法的计算方法跟计算图一样，其核心思想是通过链式求导法则，获得网络中每个参数的梯度。这里不再给出图 2-11 中的计算过程。对于神经网络的学习，如果只强调求解、推导中的数学依据对理解算法是不利的。当然，不理解数学依据更加不利，我们关心的重点不仅仅是数学本身，而更应该关注其最核心的原理，简单的高数求导是建立深度学习的基础！

到此为止，我们可以把反向传播算法视为一个黑盒子，然后跳过本节内容继续往下阅读。如果读者对反向传播算法在多层神经网络中的数学推导过程有兴趣，即



使不按照 2.4.3 节的公式推导, 也可以进一步理解反向传播算法的代码实现。

## 2. BP 算法推导

### (1) 符号说明

- $n_l$ : 表示网络的层数,  $n_1$  为输入层,  $n_L$  为输出层。
- $s_l$ : 表示第  $l$  层网络神经元的个数。
- $f$ : 表示神经元的激活函数。
- $\mathbf{w}^{(l)}$ : 表示第  $l$  层到第  $l+1$  层的权重矩阵, 其中  $w_{i,j}^{(l)} \in \mathbf{R}$  表示从  $l$  层的第  $i$  个神经元到第  $l+1$  层第  $j$  个神经元之间的权重。
- $\mathbf{b}^{(l)}$ : 表示第  $l$  层的偏置向量, 其中  $b_i^{(l)}$  表示  $l$  层第  $i$  个神经元的偏置。
- $\mathbf{z}^{(l)}$ : 表示第  $l$  层的输入向量, 其中  $z_i^{(l)}$  为  $l$  层第  $i$  个神经元的输入。
- $\mathbf{a}^{(l)}$ : 表示第  $l$  层的输出向量, 其中  $a_i^{(l)}$  为  $l$  层第  $i$  个神经元的输出。

虽然上述符号定义有些琐碎, 需要点时间去了解, 但是对于后面的公式推导却起着很重要的作用。其中,  $l$  一般代表第  $l$  层神经网络,  $i$  则一般代表当前所在第  $i$  个神经元或者输入为第  $i$  个神经元,  $j$  一般代表输出到第  $j$  个神经元上 (见图 2-12)。

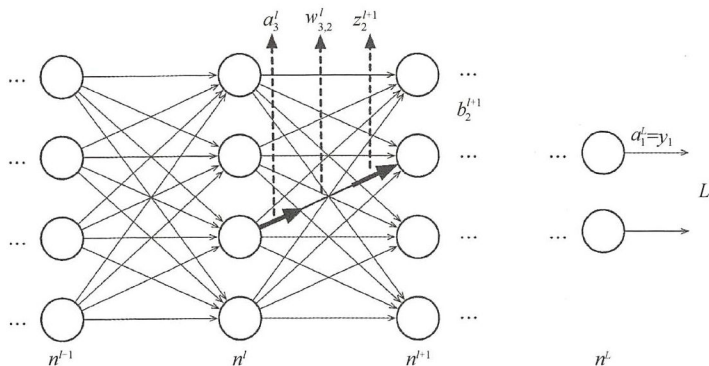


图 2-12 符号定义说明示例图

那么当前层神经网络的向前传播公式为:

$$\mathbf{a}^l = f(\mathbf{w}^{l-1} \mathbf{a}^{l-1} + \mathbf{b}^l) \quad (2-9)$$

我们把中间量  $\mathbf{w}^{l-1} \mathbf{a}^{l-1} + \mathbf{b}^l$  独立出来, 并命名为加权输入  $\mathbf{z}^l$ , 从而拆分为两个中间量公式:

$$\mathbf{z}^l = \mathbf{w}^{l-1} \mathbf{a}^{l-1} + \mathbf{b}^l \quad (2-10)$$

$$\mathbf{a}^l = f(\mathbf{z}^l) \quad (2-11)$$

在后面的反向传播算法推导的过程中, 式 (2-10) 和式 (2-11) 中的中间量非常重要。

### (2) 损失函数

假设神经网络的损失函数为  $L$ ，根据上面的符号定义有：

$$L(\mathbf{w}, \mathbf{b}) = \frac{1}{2} (\mathbf{y} - \mathbf{a}^L)^2 \quad (2-12)$$

其中， $\mathbf{y}$  为期望的输出值， $\mathbf{a}^L$  为该神经网络的预测输出值，上标  $L$  为网络的最后一层（输出层）， $1/2$  是为了方便开导时进行约减，不影响损失函数。

反向传播算法是通过改变网络中的权重参数  $\mathbf{w}$  和偏置  $\mathbf{b}$  来改变损失函数的方法，目的是计算损失函数  $L$  在神经网络中的所有权重  $\mathbf{w}$ ，以及偏置  $\mathbf{b}$  的梯度  $\partial L / \partial \mathbf{w}$  和  $\partial L / \partial \mathbf{b}$ 。

### (3) 反向传播的 4 个基本方程

首先定义第  $l$  层的第  $i$  个神经元的误差为  $\delta_i^l$ ：

$$\delta_i^l = \frac{\partial L}{\partial z_i^l} \quad (2-13)$$

反向传播算法会给出  $\delta_i^l$  的计算过程，然后利用  $\delta_i^l$  与  $\partial L / \partial w_{i,j}^l$ 、 $\partial L / \partial b_i^l$  之间的关系进行计算。表 2-1 所示是反向传播的 4 个基本方程，下面开始对其进行算法推导。

表 2-1 反向传播 4 个基本方程

方程	含义
$\delta^L = \nabla_a L \odot f'(z^L)$	BP1 输出层误差
$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l)$	BP2 第 $l$ 层误差
$\frac{\partial L}{\partial b_i^l} = \delta_i^l$	BP3 损失函数关于偏置的偏导
$\frac{\partial L}{\partial w_{i,j}^l} = a_j^{l-1} \delta_i^l$	BP4 损失函数关于权值的偏导

#### • BP1 输出层误差

根据定义， $\delta_i^L$  为输出层误差，又因为  $\mathbf{a}^L = f(\mathbf{z}^L)$ ，对损失函数求导可得：

$$\delta_i^L = \frac{\partial L}{\partial z_i^L} = \frac{\partial L}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} = \frac{\partial L}{\partial a_i^L} \frac{\partial f(z_i^L)}{\partial z_i^L} = \frac{\partial L}{\partial a_i^L} f'(z_i^L) \quad (2-14)$$

其中， $\partial L / \partial a_i^L$  度量了损失函数  $L$  在第  $i$  个神经元输出的函数的变化程度， $f'(z_i^L)$  度量了激活函数  $f(z_i^L)$  在第  $i$  个神经元加权输入处的变化程度。

式 (2-14) 以神经元为单位，对输出层误差向量化，得到输出层的误差公式，即为基本方程 BP1：

$$\text{BP1 : } \delta^L = \nabla_a L \odot f'(z^L) \quad (2-15)$$

#### • BP2 第 $l$ 层误差

非输出层的误差依赖于下一层的误差，也就是网络第  $l$  层的误差依赖于  $l+1$  层的误差。

根据式 (2-13) 有：

$$\delta_i^l = \frac{\partial L}{\partial z_i^l} = \sum_j \frac{\partial L}{\partial z_j^{l+1}} \frac{\partial z_j^{l+1}}{\partial z_i^l} = \sum_j \delta_j^{l+1} \frac{\partial z_j^{l+1}}{\partial z_i^l} \quad (2-16)$$

式 (2-15) 应用了复合函数链式求导法则，前一层的输出可以作用于后一层的输入。因为  $\frac{\partial L}{\partial z_j^{l+1}}$  相当于  $l+1$  层的误差，简化为  $\delta_j^{l+1}$ 。又因为：

式 (2-16) 应用了复合函数链式求导法则，前一层的输出可以作用于后一层的输入。因为  $\frac{\partial L}{\partial z_j^{l+1}}$  相当于  $l+1$  层的误差，简化为  $\delta_j^{l+1}$ 。又因为：

$$z_j^{l+1} = \left( \sum_i w_{i,j}^l a_i^l \right) + b_j^{l+1} = \left( \sum_i w_{i,j}^l f(z_i^l) \right) + b_j^{l+1}$$

所以有：

$$\frac{\partial z_j^{l+1}}{\partial z_i^l} = w_{i,j}^l f'(z_i^l) \quad (2-17)$$

把式 (2-17) 代入式 (2-16)，可以得到网络  $l$  层第  $i$  个神经元的误差：

$$\delta_i^l = \sum_j \delta_j^{l+1} w_{i,j}^l f'(z_i^l) \quad (2-18)$$

把式 (2-18) 向量化后可以得到网络中第  $l$  层的误差公式 BP2：

$$\text{BP2} : \delta^l = (\delta^{l+1} (w^l)^T) \odot f'(z^l) \quad (2-19)$$

公式 BP2 充分体现了误差反向传播的特点。现在结合基本方程 BP1 和 BP2，只要知道  $l+1$  层的误差，就可以间接推导得到第  $l$  层的误差。依次类推，最后能够从网络的输出层倒推直到输入层的误差。

#### • BP3 损失函数关于偏置 $b$ 的偏导

同理，根据复合函数的链式求导法则可得：

$$\frac{\partial L}{\partial b_i^l} = \sum_j \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_i^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial b_i^l} \quad (2-20)$$

接下来对第  $l$  层网络的第一个神经元的输入  $z_i^l$  关于偏置求导，其导数为 1：

$$z_i^l = \left( \sum_j w_{i,j}^{l-1} a_j^{l-1} \right) + b_i^l \Rightarrow \frac{\partial z_i^l}{\partial b_i^l} = 1 \quad (2-21)$$

因此代入式 (2-20) 可得：



$$\frac{\partial L}{\partial b_i^l} = \frac{\partial L}{\partial z_i^l} \times 1 = \frac{\partial L}{\partial z_i^l} = \delta_i^l \quad (2-22)$$

最终损失函数在网络中任意偏置 $b_i^l$ 的偏导等于该神经元上的误差。前面曾经提到过，我们需要求网络中参数的偏导，现在可以通过误差直接得到偏置的导数。

$$\text{BP3: } \frac{\partial L}{\partial b_i^l} = \delta_i^l \quad (2-23)$$

- BP4 损失函数关于权值  $w$  的偏导

损失函数关于权值  $w$  的偏导与偏置的偏导计算方法相同：

$$\frac{\partial L}{\partial w_{i,j}^l} = \sum_k \frac{\partial L}{\partial z_k^l} \frac{\partial z_k^l}{\partial w_{i,j}^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{i,j}^l} \quad (2-24)$$

$$z_i^l = \left( \sum_j w_{i,j}^{l-1} a_j^{l-1} \right) + b_i^l \Rightarrow \frac{\partial z_i^l}{\partial w_{i,j}^l} = a_j^{l-1} \quad (2-25)$$

同理，把式 (2-25) 代入式 (2-24) 中可得：

$$\text{BP4: } \frac{\partial L}{\partial w_{i,j}^l} = \frac{\partial L}{\partial z_i^l} a_j^{l-1} = a_j^{l-1} \delta_i^l \quad (2-26)$$

从公式 BP4 可以知道损失函数关于权重的偏导，连接了第  $l$  层第  $j$  个神经元的误差，以及上一层第  $i$  个神经元的输出。

在神经元传过来的激活值 $a_j^{l-1}$ 很小的情况下（如： $a \approx 0$ ），依赖梯度下降算法来更新权重将会变得很慢。也就是说，如果某个权重连接的上一个输入激活值很小，那么这个权重的学习过程就会很慢。

### 3. 反向传播算法流程

利用反向传播算法的 4 个基本方程，可以很清晰地对反向传播算法流程进行总结。

- 输入（Input）：输入层输入向量  $x$ 。
- 向前传播（Feed Forward）：计算  $z^l = w^{l-1} a^{l-1} + b^l, a^l = f(z^l)$ 。
- 输出层误差（Output Error）：根据公式 BP1 计算误差向量  $\delta^L = \nabla_a L \odot f'(z^L)$ ；
- 反向传播误差（Backpropagate Error）：根据公式 BP2 逐层反向计算每一层的误差  $\delta^l = (\delta^{l+1} (w^l)^T) \odot f'(z^l)$ 。
- 输出（Output）：根据公式 BP3 和 BP4 输出损失函数的偏置。

在训练阶段，反向传播算法的工作方式就是把数据传入神经网络，然后经过一次向前传播后，得到每一层的输出数据。接着开始从输出层往前计算每一层的误差，直到第一层（输入层）为止。最后根据每一层的误差数据计算每一个损失函数关于偏置和权重参数的偏导，而这个偏导就是网络中参数的变化率。

有了这个变化率之后，我们就可以利用梯度下降算法更新一次网络中的参数。

如此循环迭代，利用反向传播算法来求得网络参数的变化率，并用于更新网络的参数，最终使得损失函数收敛到局部最小值，训练阶段结束。

#### 4. 反向传播算法的实现

在实现反向传播算法之前，我们需要定义神经网络的模型架构，即神经网络有多少层、每一层有多少神经元。另外，还需要给出神经网络定义训练的数据和实际的输出。

在【代码清单 2-2】中，`network_sizes` 为用于测试的神经网络的模型架构，该网络一共有 3 层，输入层有 3 个神经元，隐层有 4 个神经元，输出层有 2 个神经元（如图 2-5 所示）。接着我们根据网络模型的大小，利用高斯分布函数的权重参数 `weights` 和偏置参数 `biases` 产生均值为 0、方差为 1 的随机值。下一章将会详细讲解神经网络参数的初始化方式，这里使用简单的高斯分布即可。

##### 【代码清单 2-2】反向传播算法初始化

```
# 定义神经网络的模型架构 [input, hidden, output]
network_sizes = [3, 4, 2]

# 初始化该神经网络的参数
sizes = network_sizes
num_layers = len(sizes)
biases = [np.random.randn(h, 1) for h in sizes[1:]]
weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]

# 产生训练的数据
training_x = np.random.rand(3).reshape(3, 1)
training_y = np.array([0, 1]).reshape(2, 1)
print("training data x:{}, training data y:{}\n".format(training_x, training_y))
backprop(training_x, training_y)
```

为了方便反向传播算法流程的代码实现，在【代码清单 2-3】中，我们对损失函数、激活函数和激活函数的求导进行了定义。损失函数使用了式（2-12）的均方误差作为该神经网络的损失函数，因此求导后的输入分别为网络的预测值和真实输出值，输出为两者之差。另外，激活函数使用了 Sigmoid 函数。

##### 【代码清单 2-3】反向传播算法定义损失函数和激活函数

```
def loss_der(network_y, real_y):
    """
    返回损失函数的偏导，损失函数使用 MSE
    L = 1/2(network_y-real_y)^2
    delta_L = network_y-real_y
    """
```

```

    return (network_y - real_y)

def sigmoid(z):
    """ 激活函数使用 sigmoid."""
    return 1.0 / (1.0 + np.exp(-z))

def sigmoid_der(z):
    """sigmoid函数的导数 derivative of sigmoid."""
    return sigmoid(z) * (1 - sigmoid(z))

```

完成了对神经网络模型的架构定义和一些辅助函数之后，剩下的是【代码清单 2-4】中的反向传播算法的核心实现，按照 2.4.4 节中反向传播算法流程。

`backprop()` 函数的输入为  $x$  和  $y$ ，其中  $x$  为 (3, 1) 的矩阵， $y$  为 (2, 1) 的矩阵。根据反向传播算法的 4 个基本公式 (BP1 ~ BP4) 的计算中需要知道每一层神经元的激活值和加权输入值，因此在进行向前传播时，分别使用 `activations` 记录每一层的激活值和 `zs` 记录每一层的加权输入值。

接下来是反向传播函数的核心代码，实现 4 个基本公式。既然是反向传播，那么我们首先从输出层  $L$  进行计算，因此先计算输出层的误差，然后计算损失函数关于输出层  $L$  中的偏置和权重参数的偏导。最后的循环为 `range(2, num_layers)`，意思是从网络的倒数第二层开始，往前计算第  $l$  层的损失值，以及损失函数关于第  $l$  层的偏置和权重参数的偏导。最后输出一轮反向传播后，得到关于损失函数的所有参数的偏导。

#### 【代码清单 2-4】反向传播算法的具体实现

```

def backprop(x, y):
    """
    反向传播算法实现
    """

    # 1) 初始化网络参数的导数 权重 w 的偏导和偏置 b 的偏导
    delta_w = [np.zeros(w.shape) for w in weights]
    delta_b = [np.zeros(b.shape) for b in biases]

    # 2) 向前传播 feed forward
    activation = x      # 把输入的数据作为第一次激活值
    activations = [x]   # 存储网络的激活值
    zs = []             # 存储网络的加权输入值 (z=wx+b)

    for w, b in zip(weights, biases):
        z = np.dot(w, activation) + b
        activation = sigmoid(z)

```



```
    activations.append(activation)
    zs.append(z)

# 3) 反向传播 back propagation
# BP1 计算输出层误差
delta_L = loss_der(activations[-1], y) * sigmoid_der(zs[-1])
# BP3 损失函数在输出层关于偏置的偏导
delta_b[-1] = delta_L
# BP4 损失函数在输出层关于权值的偏导
delta_w[-1] = np.dot(delta_L, activations[-2].transpose())

delta_l = delta_L
for l in range(2, num_layers):
    # BP2 计算第 l 层误差
    z = zs[-l]
    sp = sigmoid_der(z)
    delta_l = np.dot(weights[-l + 1].transpose(), delta_l) * sp
    # BP3 损失函数在 l 层关于偏置的偏导
    delta_b[-l] = delta_l
    # BP4 损失函数在 l 层关于权值的偏导
    delta_w[-l] = np.dot(delta_l, activations[-l - 1].transpose())

return (delta_w, delta_b)
```

## 2.4 示例：医疗数据诊断

IBM 在 2015 年 5 月推出 Waston Health 服务，负责收集健康数据交给 Waston 超级计算机进行医疗大数据分析。目前 IBM Waston Health 主要的应用场景在癌症病患的医疗诊疗：通过对医学影像的分析和学习，辅助医生对潜在的癌症患者进行精准诊断。

由于部分国家的医疗资源十分稀缺，提高医生的诊疗水平显得至关重要。我们的愿景并不是用人工智能取代医生帮助病人治病，医生所做的事情并不是简单的诊断，更重要的在于对诊断结果进行解释、说明和信用背书，还有针对特殊的病症进行深入治疗与跟踪。但利用人工智能、深度学习技术来辅助医生对患者进行诊断，无疑会大大提高医生的诊断水平和诊断效率，降低误诊率。现今，人工智能等技术在医疗科学中的应用已经越来越普遍了。我们会发现，医生不是直接诊断病情，而是通过 DC、彩超等手段使用机器去探测，然后通过机器分析这些数据给出



出层)，对给出不同血液检测项目 ( $x_1, x_2$ ) 的数据进行分类，判别该血液检测结果是属于病原体 I 还是病原体 II。

## 2.4.2 准备数据

首先需要导入后面所需要用到的 python 包，见【代码清单 2-5】，sklearn 用来产生例子中的数据并给出线性分类例子，numpy 是 python 处理矩阵的包，matplotlib 用于显示。

### 【代码清单 2-5】导入 python 包

```
from sklearn import linear_model
from sklearn import datasets
import sklearn
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline # 在 notebook 内显示 plt 图
```

函数 plot\_boundary 用来显示分类的边界，方便清晰地看到分类结果，见【代码清单 2-6】。输入 pred\_func 为 python 的 lambda 表达式（在这里可以理解为简单的数学函数，也就是预测函数），data 为输入的数据，labels 为分类的结果。

### 【代码清单 2-6】绘制分类边界函数

```
def plot_boundary(pred_func, data, labels):
    """ 绘制分类边界函数 """

    # 设置最大值和最小值并增加 0.5 的边界 (0.5 padding)
    x_min, x_max = data[:, 0].min() - 0.5, data[:, 0].max() + 0.5
    y_min, y_max = data[:, 1].min() - 0.5, data[:, 1].max() + 0.5
    h = 0.01 # 点阵间距

    # 生成一个点阵网格，点阵间距为 h
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # 计算分类结果 z
    z = pred_func(np.c_[xx.ravel(), yy.ravel()])
    z = z.reshape(xx.shape)

    # 绘制轮廓和训练样本，轮廓颜色使用 Blues 透明度 0.2
    plt.contourf(xx, yy, z, cmap=plt.cm.Blues, alpha=0.2)
    plt.scatter(data[:, 0], data[:, 1], s=40, c=labels, cmap=plt.
cm.Vega20c, edgecolors="Black")
```



本例中产生的医疗数据是通过 sklearn 的 datasets 产生的，并不是真实的医疗数据。见【代码清单 2-7】，这里首先需要设置 numpy 的 seed 函数，让其每次随机生成的数据都不一样。X 是产生的原始数据，y 是对应的分类，最后数据的输出结果如图 2-14 所示。

#### 【代码清单 2-7】显示医疗数据

```
np.random.seed(0)
X, y = datasets.make_moons(300, noise=0.25) # 300 个数据点，噪声设定 0.25

# 显示产生的医疗数据
plt.scatter(X[:,0], X[:,1], s = 50, c = y, cmap=plt.cm.Vega20c, edgecolors="Black")
plt.title('Medical data')
plt.show()
```

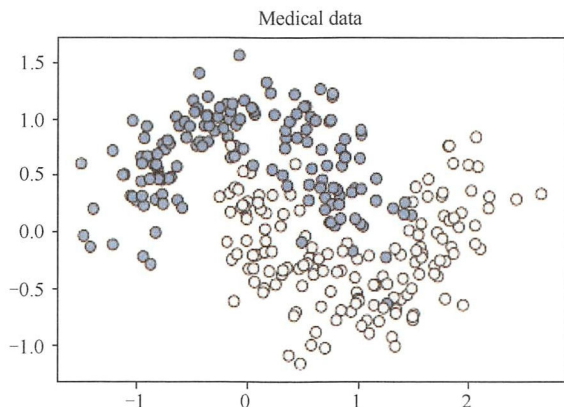


图 2-14 本例中使用 sklearn.dataset 产生医疗数据。x 轴方向表示链球菌的值，y 轴方向表示葡萄球菌的值，不同的链球菌与葡萄球菌的组合代表患者带有不同的病原体，其中深色的为病原体 I，浅色的为病原体 II

### 2.4.3 线性分类

对数据进行二分类最直接有效的方法就是采用线性回归分类器，下面使用 sklearn 的 LogisticRegressionCV 函数对图 2-14 中的球菌数据进行二分类，见【代码清单 2-8】。

#### 【代码清单 2-8】对球菌数据进行线性二分类

```
# 使用 scikit-learn 的线性回归分类器
logistic_fun = sklearn.linear_model.LogisticRegressionCV()
```

```
logistic_fun.fit(X, y)

# 显示线性分类结果
plot_decision_boundary(lambda x: clf.predict(x), X, y)
plt.title("Logistic Regression")
```

图 2-15 所示为对球菌数据进行逻辑回归分类的结果。逻辑回归算法直接用直线将数据分隔为两类。从图中可以清晰地看到该分类结果远远不能满足真实目的，因为约有 30% 的数据分类错误，对医疗诊断结果来说，该精度是不可以接受的。

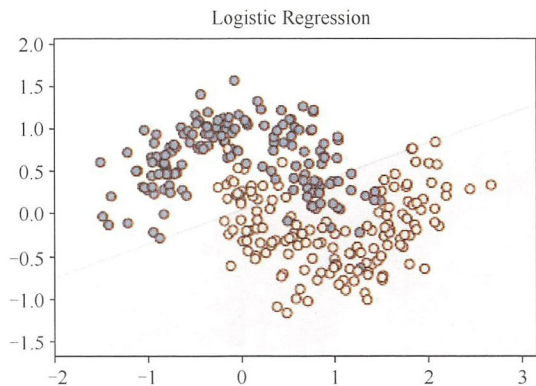


图 2-15 使用线性分类器对医疗数据进行分类

在现实生活当中，数据大多是线性不可分的，因此我们不能简单地使用直线将数据划分为两份。这意味着不能用线性分类器来分类，例如 Logistic 分类器、贝叶斯分类器等，否则分类后数据精度和召回率会过低，对真实的医疗诊断结果造成巨大的影响。为了减少误差，提升分类的结果和质量，接下来采用神经网络对该医疗数据进行预测分类。

## 2.4.4 建立人工神经网络模型

为了解决例子中医疗数据的线性不可分问题，我们在这里建立一个 3 层神经网络模型，包括输入层、隐层、输出层。

输入层中的神经节点数目由输入数据的维数决定，本例中有链球菌  $x_1$  和葡萄球菌  $x_2$ ，因此输入数据维度是 2，输入由 2 个神经元节点组成。同理，输出层中的节点数目由输出的分类维度决定，本例中为病原体 I(0) 和病原体 II(1)，因此输出层由 2 个神经元组成。最后该人工神经网络的定义如图 2-16 所示。

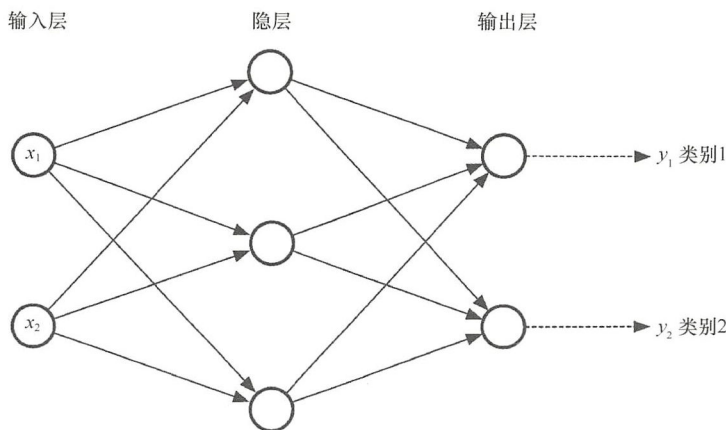


图 2-16 使用到的 3 层人工神经网络，其中隐层为 3 个神经节点

根据实际情况，我们可以增加隐层数和每一隐层的节点数，隐层越多，隐层节点越多，就能够处理更加复杂数据模型，但是代价是开销更大：

- 隐层越多，意味着网络模型越大，引入的权值参数越多，占用的 GPU 显存越多；
- 参数数量越多，数据过度拟合的可能性就越大，网络越有可能不稳定，预测效果反而下降。

那么如何选择适当的隐层数呢？每层隐藏层的神经元数应该设置为多少呢？这是一个特征工程问题，不同的数据会有不一样的结果。所以神经网络的设计是一种工程艺术，我们需要去尝试、训练、预测、评估，才能最终确定整个网络模型的形态。

隐层需要一个激活函数，激活函数把输出层转换为下一层的输入层。非线性的激活函数能够让我们去处理非线性的问题。常用的非线性激活函数有 Tanh 函数、Sigmoid 函数和 ReLU 函数。本例中我们选择使用 Tanh 函数，同样也可以尝试把 Tanh 函数换成其他函数查看输出，最后通过 Softmax 层把激活函数的输出转换为概率。

### Softmax 层

在神经网络中 Softmax 函数常常作用于输出层，将神经网络的输出向量转换成同分布的概率分布。其在神经网络结构中所处的位置如图 2-17 所示，在第 3 章中将会对 Softmax 层进行详细解释。



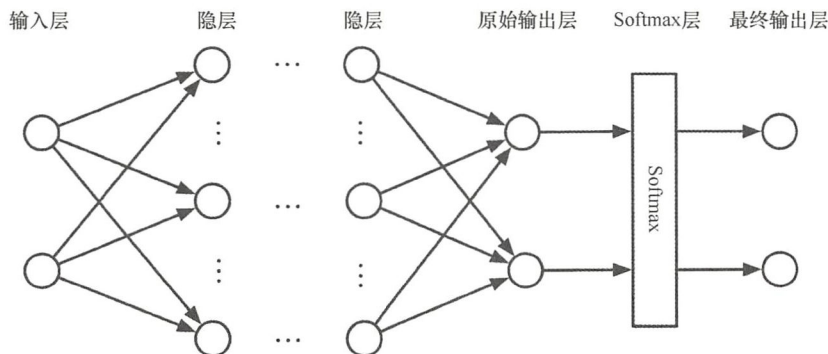


图 2-17 Softmax 层所处的位置示例图

## 1. 网络定义

人工神经网络的预测使用向前传播操作，通常可以理解为激活函数和矩阵乘法的操作。假设输入  $x$  是二维数据，根据神经网络的定义，其网络输出分类结果为  $\hat{y}$ 。根据图 2-16 的神经网络模型有：

$$h = \tanh(xW_1 + b_1)$$

$$\hat{y} = \text{softmax}(hW_2 + b_2)$$

其中， $x$  为神经网络的输入数据， $h$  为隐层的输出， $\hat{y}$  为输出层的输出。激活函数使用  $\tanh()$ ，输出分类器使用  $\text{softmax}()$ 。为了方便代码的编写和反向传播的公式计算，现对上面两个公式进行展开：

$$z_1 = xW_1 + b_1$$

$$h_1 = \tanh(z_1)$$

$$z_2 = h_1W_2 + b_2$$

$$h_2 = \hat{y} = \text{softmax}(z_2)$$

$z_i$  是第  $i$  层的输入（隐层的输入）， $h_i$  是第  $i$  层激活函数处理后的输出（隐层的输入）。 $W_1$ 、 $b_1$ 、 $W_2$ 、 $b_2$  是神经网络的权重参数和偏置参数，在网络训练时会使用到。在神经网络中，通常使用矩阵来表达参数  $W_i$ 、 $b_i$ ，以方便后续存储和计算。

如果隐层中使用 100 个神经元节点，那么有  $W_1 \in \mathbf{R}^{2 \times 100}$ 、 $b_1 \in \mathbf{R}^{100}$ 、 $W_2 \in \mathbf{R}^{100 \times 2}$ 、 $b_2 \in \mathbf{R}^2$ 。从上面参数的计算公式可以看出，增加隐层的节点数会大量的增加神经网络模型中的参数，也就是网络中的参数  $W_i$ 、 $b_i$  随着神经元或者隐层的增加而急剧增加。

如果隐层中使用 100 个神经元节点，那么有  $W_1 \in \mathbf{R}^{2 \times 100}$ 、 $b_1 \in \mathbf{R}^{100}$ 、 $W_2 \in \mathbf{R}^{100 \times 2}$ 、 $b_2 \in \mathbf{R}^2$ 。从上面参数的计算公式中可以看出，增加隐层的节点数会使神经网络模型中的参数大量增加，也就是网络中的参数  $W_i$  和  $b_i$  会随着神经元或者隐层数目的增加

而急剧增加。

## 2. 计算网络参数

对神经网络进行训练，也就是要找到神经网络中的参数  $W_i$  和  $b_i$ ，使得输出的数据与实际输出数据的误差最小。前面曾经提到过检测误差的函数叫作损失函数（Loss Function），本例中使用常用交叉熵损失函数（也叫作负对数似然函数）。如果我们有  $N$  个训练样本，对应  $C$  个分类，那么预测值  $\hat{y}$  与实际值  $y$  之间的损失函数为：

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log \hat{y}_{n,i}$$

实际值  $y$  与预测值  $\hat{y}$  之间的概率分布差别越大，损失就越大。因此通过寻找一个较优的权重参数，能够最大限度地减少损失，提高数据分类的准确性。

本例中使用梯度下降算法来寻找损失函数的最小值。为了方便编码和实现，此处使用一个固定的学习率  $\eta$  实现批量梯度下降算法（在实际工程中，通常是使用随机梯度下降算法或小批量随机梯度下降算法）：

$$x \leftarrow x - \eta \Delta x$$

上式中  $\Delta x$  为对  $x$  进行求导，因此批量梯度下降首先需要对该人工神经网络模型中用到的参数  $W_1$ 、 $b_1$ 、 $W_2$ 、 $b_2$  进行求导：

$$\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2}$$

对权重和偏置参数进行求导则是利用 BP 反向传播算法（此处直接使用 2.2.3 节中 BP 算法推导得到的 4 个反向传播基本方程），得到网络中各个参数的偏导为：

$$\begin{aligned} \delta_3 &= \hat{y} - y \\ \delta_2 &= (1 - \tanh^2 z_1) \times \delta_3 W_2^T \\ \frac{\partial L}{\partial W_2} &= a_1^T \delta_3 \\ \frac{\partial L}{\partial b_2} &= \delta_3 \\ \frac{\partial L}{\partial W_1} &= x^T \delta_2 \\ \frac{\partial L}{\partial b_1} &= \delta_2 \end{aligned}$$

## 3. Python 实现

现在我们已经得到该神经网络模型中损失函数相关的参数梯度计算公式，并定义了神经网络的参数。下面需要定义梯度下降算法中的参数，如【代码清单 2-9】所示。

## 【代码清单 2-9】定义参数

```
input_dim = 2      # 输入的维度
output_dim = 2     # 输出的维度, 分类数

epsilon = 0.01     # 梯度下降算法的学习率
reg_lambda = 0.01  # 正则化强度
```

下面来实现 `calculate_loss()` 损失函数, 如【代码清单 2-10】所示, `model` 字典中存储着网络中的权重参数和 偏置参数, 通过 `model` 获取网络中的参数  $W_1$ 、 $b_1$ 、 $W_2$ 、 $b_2$ , 使用神经网络的前馈公式进行前馈计算。前馈计算后得到预测的输出值 `probs`, 再用 `probs` 与真实的输出值  $y$  来计算损失值。

## 【代码清单 2-10】损失函数

```
def calculate_loss(model, X, y):
    ''' 损失函数 '''

    num_examples = len(X)  # 训练集大小
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']

    # 使用正向传播计算预测值
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

    # 计算损失值
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)

    # 对损失值进行归一化
    data_loss += reg_lambda / 2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1. / num_examples * data_loss
```

如【代码清单 2-11】所示, 对于预测函数 `predict()`, 预测时只需要对模型进行一次向前传播, 然后返回分类结果中概率值最大的一项。

## 【代码清单 2-11】预测函数

```
def predict(model, x):
    ''' 预测函数 '''

    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
```

```

# 向前传播
z1 = x.dot(W1) + b1
a1 = np.tanh(z1)
z2 = a1.dot(W2) + b2
exp_scores = np.exp(z2)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
return np.argmax(probs, axis=1)

```

【代码清单 2-12】为该人工神经网络模型的函数，ANN\_model() 函数实现了反向传播算法来计算梯度下降，计算公式使用了 2.3.2 节中参数学习的公式。

### 【代码清单 2-12】人工神经网络模型整体函数

```

def ANN_model(X, y, nn_hdim):
    """
    人工神经网络模型函数
    - nn_hdim: 隐层的神经元节点（隐层的数目）
    """
    num_indim = len(X) # 用于训练网络的输入数据
    model = {} # 模型存储定义

    # 随机初始化网络中的权重参数 w1、w2 和偏置 b1、b2
    np.random.seed(0)
    W1 = np.random.randn(input_dim, nn_hdim) / np.sqrt(input_dim)
    b1 = np.zeros((1, nn_hdim))
    W2 = np.random.randn(nn_hdim, output_dim) / np.sqrt(nn_hdim)
    b2 = np.zeros((1, output_dim))

    # 批量梯度下降算法 BSGD
    num_passes=20000 # 梯度下降迭代次数
    for i in xrange(0, num_passes):
        # 向前传播
        z1 = X.dot(W1) + b1 # M[200*2] * M[2*3] --> M[200*3]
        a1 = np.tanh(z1)
        z2 = a1.dot(W2) + b2 # M[200*3] * M[3*2] --> M[200*2]
        exp_scores = np.exp(z2)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        # 向后传播算法
        delta3 = probs # 得到的预测值
        delta3[range(num_indim), y] -= 1 # 预测值减去实际值
        delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
        dW2 = (a1.T).dot(delta3) # W2 的导数
        db2 = np.sum(delta3, axis=0, keepdims=True) # b2 的导数
        dW1 = np.dot(X.T, delta2) # W1 的导数

```



```

db1 = np.sum(delta2, axis=0)          # b1 的导数

# 添加正则化项
dW1 += reg_lambda * W1
dW2 += reg_lambda * W2

# 根据梯度下降算法更新权重
W1 += -epsilon * dW1
b1 += -epsilon * db1
W2 += -epsilon * dW2
b2 += -epsilon * db2

# 把新的参数写入 model 字典中进行记录
model = {'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}

if i % 1000 == 0:
    print("Loss after iteration %i: %f" % i, calculate_loss(model, X, y))

return model

```

到此为止，我们可以运行上面的人工神经网络的模型，如【代码清单 2-13】所示。其中使用到了 Python 的 lambda 表达式，lambda x: predict(hidden\_3\_model, x) 中的 hidden\_3\_model 为经过 ANN 网络迭代训练 20000 次后的权重参数字典，其中存储了人工神经网络模型用到的参数  $W_1$ 、 $b_1$ 、 $W_2$ 、 $b_2$ 。lambda 表达式是为了把 predict() 函数连带输入参数直接传入 plot\_decision\_boundary() 函数中，然后在 plot\_decision\_boundary() 函数需要执行 predict() 时才执行，这是 Python 的神奇函数执行方法。

#### 【代码清单 2-13】运行该人工神经网络模型代码

```

hidden_3_model = ANN_model(X, y, 3) # 建立 3 个神经元的隐层
plot_decision_boundary(lambda x: predict(hidden_3_model, x), X, y)
plt.title("Hidden Layer size 3")

```

下面来看只有 3 个隐层节点数的情况，该网络模型训练结果的输出如【代码清单 2-14】所示，对输入的数据进行预测分类的结果如图 2-18 所示。

#### 【代码清单 2-14】网络模型训练阶段输出

```

>>> Loss after iteration 0: 0.389105
>>> Loss after iteration 1000: 0.120831
>>> Loss after iteration 2000: 0.116688
...
>>> Loss after iteration 18000: 0.114781
>>> Loss after iteration 19000: 0.114780
>>> Loss after iteration 20000: 0.114779

```

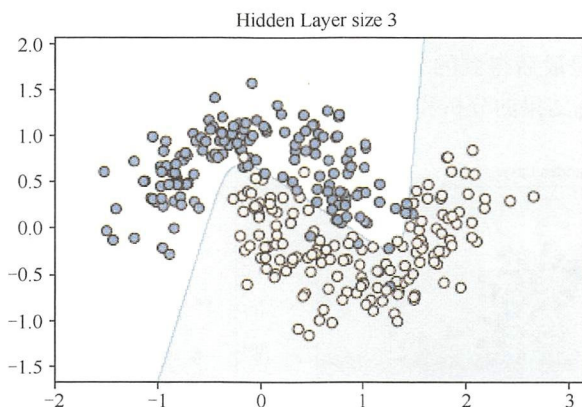


图 2-18 使用 ANN 对本例中的医疗数据进行分类，隐层节点数为 3

## 2.4.5 隐层节点数对人工神经网络模型的影响

对比线性分类的结果（图 2-15）和 ANN 模型分类的结果（图 2-18），可以看出 ANN 模型对该医疗数据的预测效果比简单地使用 Logistic 回归进行分类的效果要好很多，但是仍然存在少量数据分类错误。

接下来探讨不同的隐层节点数（隐层中神经元的数量）对 ANN 模型的影响。【代码清单 2-15】使用 list 来记录被测试的不同隐层节点数，分别是 [1, 2, 3, 4, 5, 15, 30, 50]，然后使用 for 来迭代 Python 的 enumerate 对象。

### 【代码清单 2-15】不同隐层节点对模型的影响

```
# 定义输出图像的大小
plt.figure(figsize=(16, 32))

# 待输入的隐层节点数 list
hidden_layer_dimensions = [1, 2, 3, 4, 30, 50]

for i, nn_hdim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer size %d' % nn_hdim)
    model = ANN_model(X, y, nn_hdim) # 运行 ANN 模型
    plot_decision_boundary(lambda x: predict(model, x), X, y) # 输出 ANN 模型分类结果

plt.show()
```

如图 2-19 所示，隐层数在低维（3、4、5）时能够很好地表达数据的分类属性。随着隐层中节点数的增加，过度拟合的可能性越高（如隐层数为 50）。本例表明，神

神经网络并不是隐层数越多或者隐层节点越多效果越好，理论上神经网络可以模拟任何形式的函数，但最后得到的分类空间可能并不是我们想要得到的效果。因此在实际工程中，对训练结果的分析和可视化就显得尤为重要。

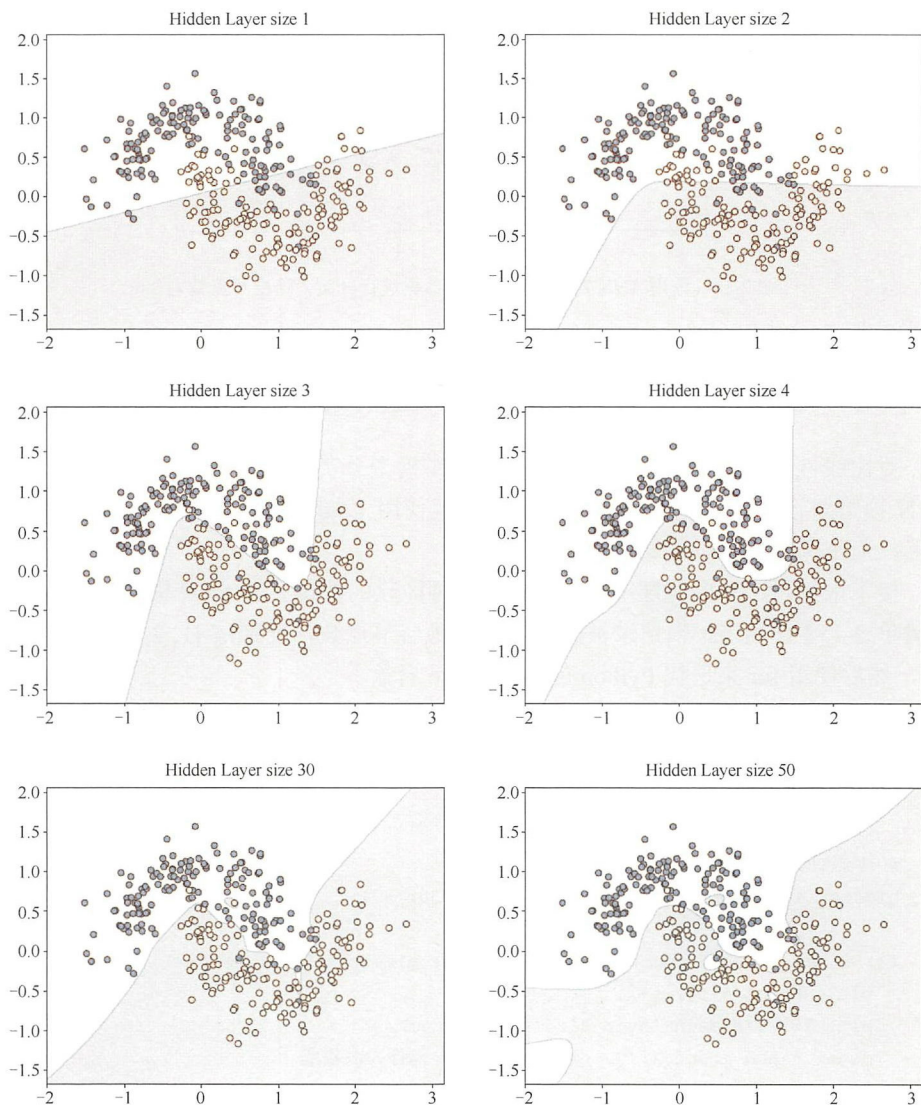


图 2-19 不同隐层节点数对人工神经网络 ANN 模型的输出结果影响，隐层节点数分别为 1、2、3、4、30、50，该人工神经网络 ANN 模型使用 3 层结构

上述例子使用了医疗血液检测数据对人体杆菌进行分类检测，但实际上，由于

医疗数据规模小、标注质量差、了解算法的人员不具备医疗背景，因此临床医疗上的分析和报告仍然由需要医生对大量的医疗数据的质量进行严格把关。

即使目前深度学习在某些医学征象的识别方面取得了大量的进展，但无论是在国内还是国外，都没有开始大规模地将深度学习应用于医疗影像上的临床验证，因为仍缺乏验证的技术和科学方法。深度学习训练得到的网络模型属于一个黑箱子，大多数情况下无法对预测的数据给出合理的解释。

不过，国内外医学界的多位著名专家表示：来自临床的需求是医学发展的动力。因此我们不必对深度学习在医疗方面的应用过于悲观，而应该保持积极的心态继续用科技去改变世界！

## 2.5 本章小结

人工神经网络是深度学习的基石。本章作为开篇的知识章节，首先回顾了人工神经网络发展历程中的 3 次高潮，每一次高潮都给神经网络带来了更多的想象空间。在对人工神经元的介绍中，我们首先从生物神经元的角度看神经元模型，然后对神经元模型进行数学解释。有了神经元模型后，对神经元进行组合排列，就可以得到多层神经网络。

多层神经网络是深度学习后续发展的基础，其分为训练阶段和预测阶段。在训练阶段，我们通过梯度下降算法更新网络中的权重参数和偏置参数。梯度下降算法需要求得网络参数的梯度，由于神经网络的结构特殊性，我们引入了反向传播算法对网络参数梯度进行求解。在预测阶段，使用向前传播算法对新输入的数据进行预测。

- 神经元模型： $y = f(W^T X + b)$ 。一个神经元的基本功能是对输入向量  $X$  与权值向量  $W$  内积求和后，经过非线性的激活函数  $f$ ，得到  $y$  作为输出结果。
- 神经元的线性模型：单个神经元模型在没有加入激活函数之前，可以看作一个线性回归模型，即把输入的数据映射到一个  $n$  维的平面空间中。
- 激活函数的主要作用是为该神经元引入了非线性因素，从而使神经元模型可以更好地解决复杂的数据分布，而不仅限于处理线性数据任务。
- 人工神经网络的组成包括输入层、隐层、输出层。其中，隐层由多个神经元组合而成，一个神经网络模型中可以有多层隐层。
- 梯度下降算法：沿着与梯度向量相反的方向，梯度下降最快，易于找到函数的最小值。因此，梯度下降的目标就是找到参数  $\theta^*$ ，使得神经网络总损失  $L$  最小，从而确定神经网络中的权重向量  $W$  和偏置  $b$ 。



## 引用/参考

- [1] McCulloch W S. A logical calculus of ideas imminent in nervous activity[J]. Biol Math Biophys, 1943, 5.
- [2] Kleene S C. Representation of events in nerve nets and finite automata[J]. Automata Studies Annals of Mathematics Studies, 1956:3-41.
- [3] Donald O. The Organization of Behavior[J]. 1949, 9(3):213-218.
- [4] Farley B, Clark W. Simulation of self-organizing systems by digital computer[J]. Transactions of the Ire Professional Group on Information Theory, 2003, 4(4):76-84.
- [5] Rochester N, Holland J H, Haibt L H, et al. Tests on a cell assembly theory of the action of the brain, using a large digital computer. IRE Trans 2:80-93[J]. Information Theory Ire Transactions on, 1956, 2(3):80-93.
- [6] Rosenblatt F. The perceptron: a probabilistic model for information storage and organization in the brain.[M]// Neurocomputing: foundations of research. MIT Press, 1988:386-408.
- [7] Werbos P. Beyond Regression : New Tools for Prediction and Analysis in the Behavioral Science[J]. Ph.d.dissertation Harvard University, 1974, 29(18):65-78.
- [8] Hubel D H, Wiesel T N. Brain and visual perception: The story of a 25-year collaboration.[J]. Color Research & Application, 2010, 31(2):156-156.
- [9] Schmidhuber J. Deep learning in neural networks: An overview[J]. Neural Netw, 2014, 61:85-117.
- [10] Ivakhnenko A G, Lapa V G. CYBERNETIC PREDICTING DEVICES,[J]. Transdex, 1966.
- [11] Ivakhnenko A G, Lapa V G, Scripta Technica I, et al. Cybernetics and forecasting techniques[M]. American Elsevier, 1967.
- [12] Casper M, Mengel M, Fuhrmann C, et al. Perceptrons: An Introduction to Computational Geometry[J]. 1972, 75(3):3356-62.
- [13] Rumelhart D E, McClelland J L. in Parallel Distributed Processing. Explorations in the Microstructure of Cognition[J]. Language, 1968, 63(4).
- [14] Weng J, Ahuja N, Huang T S. Cresceptron: a self-organizing neural network which grows adaptively[C]// International Joint Conference on Neural Networks. IEEE Xplore, 1992:576-581 vol.1.
- [15] Weng J, Ahuja N, Huang T S. Learning recognition and segmentation using the Cresceptron[J]. International Journal of Computer Vision, 1997, 25(2):109-143.
- [16] Hochreiter S. Untersuchungen zu dynamischen neuronalen Netzen[C]// Master' s Thesis, Institut Fur Informatik, Technische Universitat, Munchen. 1991.
- [17] Kolen J F, Kremer S C. Gradient Flow in Recurrent Nets: The Difficulty of Learning Long Term

- Dependencies[C]// Wiley-IEEE Press, 2001:237-243.
- [18] Schmidhuber J, rg. Learning complex, extended sequences using the principle of history compression[M]. MIT Press, 1992.
- [19] Behnke S. Hierarchical Neural Networks for Image Interpretation[J]. Lecture Notes in Computer Science, 2003, 2766(3):1345-1346.
- [20] Smolensky. Information processing in dynamical systems: foundations of harmony theory[C]// MIT Press, 1986:194-281.
- [21] Hinton G E. Deep belief networks[J]. Scholarpedia, 2009, 4(6):5947.
- [22] Le Q V. Building high-level features using large scale unsupervised learning[C]// IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 2013:8595-8598.
- [23] Yang J J, Pickett M D, Li X, et al. Memristive switching mechanism for metal/oxide/metal nanodevices[J]. Nature Nanotechnology, 2008, 3(7):429-433.
- [24] Nielsen, Michael A. Neural Networks and Deep Learning[M]// Machine Learning. Elsevier Ltd, 2015.

---

# 第 3 章

---

## 深度学习基础 及技巧

本章主要内容：

- 激活函数（Activation Function）
- 损失函数（Loss Function）
- 超参数（Hyper Parameters）
- 深度学习的技巧

在上一章中，我们学习了人工神经网络 ANN，明确了人工神经网络 ANN 的模型由输入层、隐层、输出层组成。准备好需要训练和测试的数据后，我们可以自定义一个神经网络模型，其中包括输入层、输出层的节点数、隐层数量。最后通过定义合理的损失函数，模型根据反向传播算法（BP）和随机梯度下降算法（SGD），自动去修正神经网络中的参数，并对训练数据的特征进行学习。

虽然我们对人工神经网络 ANN 有了初步的了解，还实现了一个简单的三层神经网络来对虚拟医疗数据进行分类。可在中间还忽略了很多细节，例如有没有其他的激活函数可供选择？损失函数只能使用均方误差（MSE）吗？如何确定梯度下降算法的参数？除了上述数学问题外，还有如何解决训练时模型预测的准确率不再提升的问题？当隐层神经元设置过多引起过度拟合时，如何通过减少过度拟合的情况来提高网络模型的预测准确率？

不懂机器学习数学原理的算法工程师，并不能称为真正的算法工程师。不懂神经网络数学原理的算法工程师，并不是真正的深度学习拥护者。可毕竟“条条大路通罗马”，总有方法可以让我们少走弯路。在本章中，我们将会深入了解神经网络中激活函数、损失函数的定义和类型。此外，在训练阶段使用的梯度下降算法中，常常需要调节与神经网络相关的众多参数（对网络模型进行参数调整的过程称为“调参过程”），恰当地定义好参数不仅可以加快网络模型的训练速度，还能够提升训练效果，起到事半功倍的作用。在本章结束前，我们将会讨论深度学习的一些规律和技巧。

如果发现自己训练的神经网络模型效果一般，可以回顾本章，尝试使用深度神经网络中的一些小技巧来提升网络模型的效果。

## 3.1 激活函数

在正式了解神经网络的激活函数（Activation Functions）之前，我们首先思考几个问题：为什么需要激活函数？激活函数可以有哪些种类？如何选择激活函数？下面带着这几个问题来深入了解神经网络中的激活函数。

首先来看激活函数在神经网络模型中的位置，如图 3-1（a）所示的激活函数实际上隐藏在神经网络模型中的连接线上。前文反复强调神经网络模型中的连接是最重要的，该连接线包括输入矩阵与权值矩阵相乘的过程，还包括激活函数对数据进行激活的过程。

激活函数的一般性质如下。

### （1）单调可微

一般情况下，我们使用梯度下降算法更新神经网络中的参数，因此必须要求激



活函数可微。如果函数是单调递增的，求导后函数必大于零（方便计算），因此需要激活函数具有单调性。

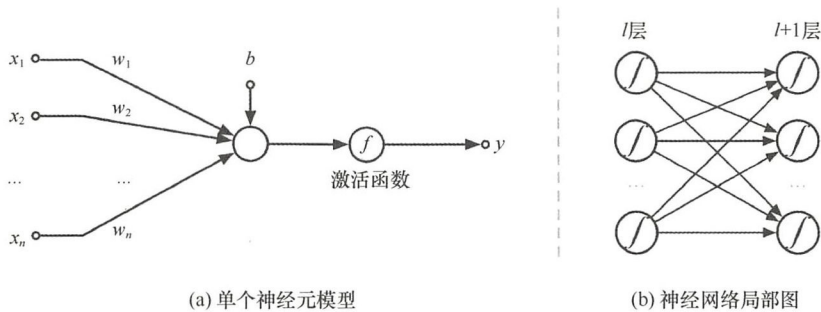


图 3-1 激活函数作用于  $l$  层的神经元到  $l+1$  层神经元之间。其中 (a) 为单个神经元模型，激活函数用  $f$  表示；(b) 为神经网络的第  $l$  层和第  $l+1$  层，每个圆圈表示一个激活函数

(2) 限制输出值的范围

输入的数据通过神经元上的激活函数来控制输出数值的大小，该输出数值是一个非线性值。通过激活函数求得的数值，根据极限值来判断是否需要激活该神经元。也就是说，我们可以通过激活函数确定是否对一个神经元的输出感兴趣。

(3) 非线性

因为线性模型的表达能力不够（从数据输入到与权值求和加偏置，都是在线性函数内执行权重与输入数据进行加权求和的过程，如  $z = WX + b$ ），因此激活函数的出现还为神经网络模型加入非线性因素。

激活函数拥有上述特性，它存在的核心意义在于：一个没有激活函数的神经网络只不过是一个线性回归模型，它不能表达复杂的数据分布。神经网络中加入了激活函数，相当于引入了非线性因素，从而解决了线性模型不能解决的问题。

选择不同的激活函数对神经网络的训练和预测都有着不同程度的影响，下面来分析神经网络中常用的激活函数及其优缺点。

### 3.1.1 线性函数

如图 3-2 所示，线性（Linear）函数是最基本的激活函数，其因变量与自变量之间有直接的比例关系。因此，线性变换类似于线性回归。实际上，在神经网络中线性变换，意味着节点按原样通过数据信号通常与单层神经网络一起使用。线性激活函数如式 (3-1) 所示，代码见【代码清单 3-1】。

$$f(x)=ax+b \quad (3-1)$$

### 【代码清单 3-1】线性激活函数

```
def linear(x, a, b):
    return a * x + b
```

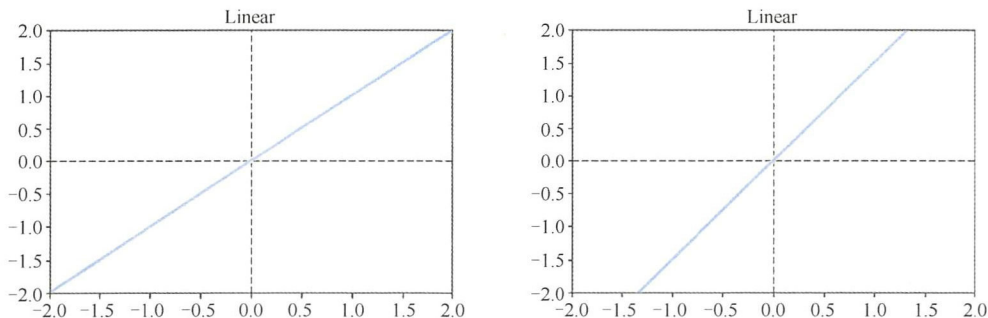


图 3-2 线性函数

## 3.1.2 Sigmoid函数

如图 3-3 所示, Sigmoid 函数是一种在不删除数据的情况下, 减少数据的极值或异常值的函数。因此, 它能够很好地表达激活的意思, 未激活值为 0, 完全饱和的激活值则为 1。在数学上, Sigmoid 激活函数为每个类输出提供独立的概率。Sigmoid 激活函数如式 (3-2) 所示, 代码见【代码清单 3-2】。

$$s(x) = \frac{1}{1 + e^{-ax}} \quad (3-2)$$

优点:

- Sigmoid 函数的输出映射在  $[0, 1]$  范围内, 函数单调连续, 且输出范围有限制, 优化稳定;
- 易于求导;
- 输出值为独立概率, 可以用在输出层。

缺点:

- Sigmoid 函数容易饱和, 导致训练结果不佳;
- 其输出并不是零均值, 数据存在偏差, 分布不平均。

### 【代码清单 3-2】Sigmoid 激活函数

```
def sigmoid(x, w = 1):
    return 1 / (1 + np.sum(np.exp(-wx)))
```

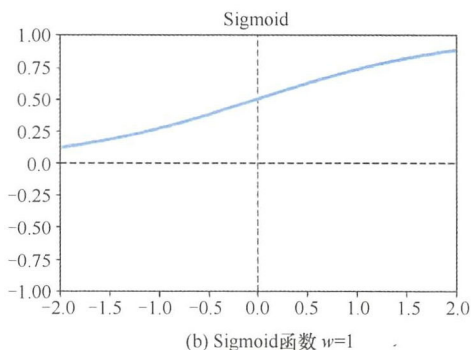
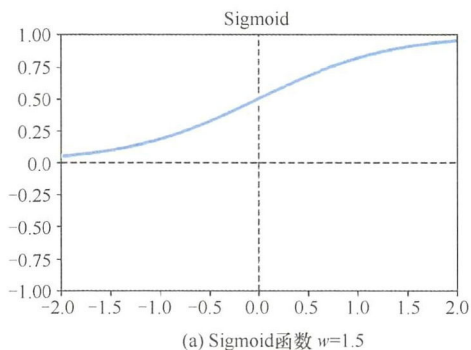


图 3-3 Sigmoid 函数

### 3.1.3 双曲正切函数

如图 3-4 所示，双曲正切函数 (Tanh) 与 Sigmoid 函数类似。不同的是，Tanh 的归一化范围为 -1 到 1，而不是 0 到 1，因此 Tanh 的优点是可以更容易地处理负数。同时，因为 Tanh 是 0 均值，也就解决了 Sigmoid 函数的非 0 均值的缺点，所以实际中 Tanh 函数会比 Sigmoid 函数更常用。Tanh 激活函数如式 (3-3) 所示，代码见【代码清单 3-3】。

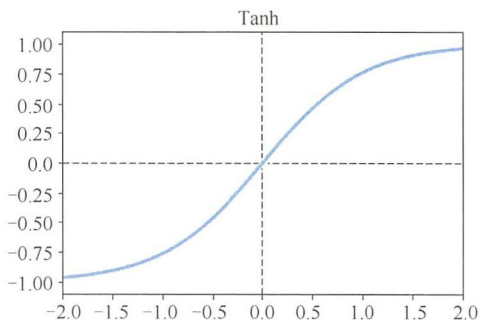


图 3-4 双曲正切函数

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (3-3)$$

优点：

- Tanh 函数比 Sigmoid 函数收敛速度更快，更加易于训练；
- 其输出以 0 为中心，数据分布平均。

缺点：

- 没有改变 Sigmoid 函数由饱和性引起的梯度消失问题。

【代码清单 3-3】双曲正切激活函数

```
def tanh(x):
    return np.tanh(x)
```

### 3.1.4 ReLU函数

(Lennie et al., 2003) 的研究表明, 大脑同时被激活的神经元只有 1% ~ 4%, 这表明神经元工作的稀疏性。从信号方面来看, 神经元同时只对输入信号的少部分进行选择响应, 大量信号被刻意地屏蔽, 这样可以提高神经网络的学习精度, 更好地提取稀疏特征。而 ReLU 函数 (如图 3-5 所示) 则满足仿生学中的稀疏性, 只有当输入值高于一定数目时才激活该神经元节点。当输入值低于 0 时进行限制, 当输入值上升到某一阈值以上时, 函数中的自变量与因变量呈线性关系。ReLU 激活函数如式 (3-4) 所示, 代码见【代码清单 3-4】。

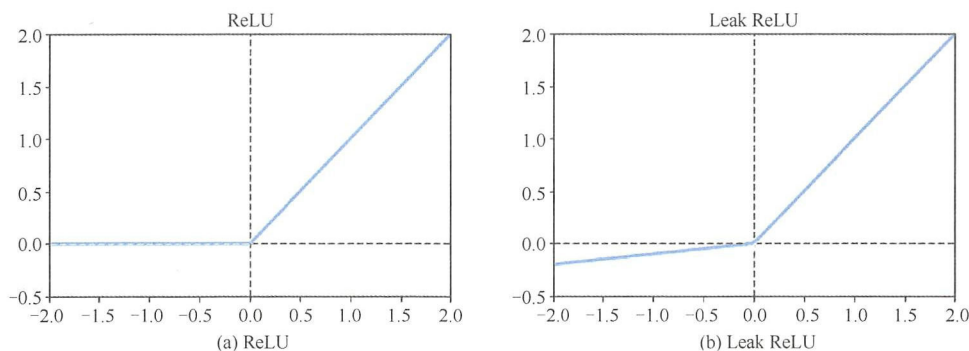


图 3-5 ReLU 类函数

$$\text{relu}(x) = \max(0, x) \quad (3-4)$$

ReLU 函数对比 Sigmoid 函数主要有如下 3 点变化:

- 单侧抑制;
- 相对宽阔的兴奋边界;
- 稀疏激活性。

优点:

- 相比 Sigmoid 函数和 Tanh 函数, ReLU 函数在随机梯度下降算法中能够快速收敛;
- ReLU 函数的梯度为 0 或常数, 因此可以缓解梯度消散问题;
- ReLU 函数引入稀疏激活性, 在无监督预训练时也能有较好的表现。

缺点:

- ReLU 神经元在训练中不可逆地死亡;
- 随着训练的进行, 可能会出现神经元死亡、权重无法更新的现象, 流经神经元的梯度从该点开始将永远是零。



正是因为 ReLU 函数比其他激活函数更适合在神经网络中作为激活函数，或者说优点更加明显，因此综合速率和效率，神经网络中大部分激活函数都使用了 ReLU 函数。

#### 【代码清单 3-4】ReLU 激活函数

```
def relu(x):
    return x if x > 0 else 0
```

### 3.1.5 Softmax函数

根据维基百科对 Softmax 函数的定义：Softmax 函数的本质是将一个  $K$  维的任意实数向量，压缩（映射）成另一个  $K$  维的实数向量，其中向量中的每个元素取值都介于  $(0, 1)$  范围内。

Softmax 是对逻辑回归（Logistic Regression, LR）的推广，逻辑回归用于处理二分类问题，其推广 Softmax 回归则用于处理多分类问题。如图 3-6 所示，在数学上，Softmax 函数会返回输出类的互斥概率分布，例如，网络的输出为  $(1, 1, 1, 1)$ ，经过 Softmax 函数后输出为  $(0.25, 0.25, 0.25, 0.25)$ 。我们可以得到分类中唯一所属类别，因此通常把 Softmax 作为输出层的激活函数。Softmax 激活函数如式 (3-5) 所示，见【代码清单 3-5】。

$$\text{softmax}(x_j) = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}} \quad (3-5)$$

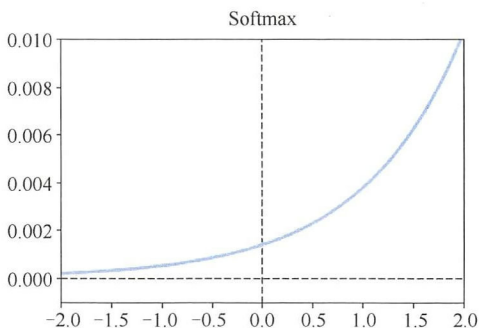


图 3-6 Softmax 函数曲线图

下面进一步说明 Softmax 函数作为输出层的思路及其使用方法。现假设有一个多分类问题，但是我们只关心这些类别的最高得分概率，那么将会使用一个带有最大似然估计函数的 Softmax 输出层来获得所有类别输出概率的最大值。例如神经网络的

分类有 3 个，分别为“野马”“河马”“斑马”，使用 Softmax 作为输出层的激活函数最后只能得到一个最大的分类概率如：野马（0.6）、河马（0.1）、斑马（0.3），其中最大值为野马（0.6）。

如果要求每次的输出都可以获得多个分类，例如希望神经网络的预测输出既像“野马”也像“河马”，那么我们不希望 Softmax 作为输出层，这里可以使用 Sigmoid 函数作为输出层的激活函数更合适，因为 Sigmoid 函数可以为每个类别的输出提供独立的概率。

#### 【代码清单 3-5】Softmax 激活函数

```
def Softmax(x):  
    return np.exp(x) / np.sum(np.exp(x))
```

### 3.1.6 激活函数的选择

在选择激活函数时，一般隐层选择 Leak ReLU 函数会得到较为理想的效果。当然这不是恒定的规律，我们可以尝试使用 Sigmoid 函数作为隐层激活函数，但注意使用时尽量不要超过太多隐层。另外可以使用 Tanh 函数来代替 Sigmoid 函数观察模型的精确率曲线图。如果直接使用 ReLU 函数作为激活函数，注意梯度下降算法的学习率参数  $\eta$  不能设置得过高，避免神经元的大量“消亡”。对于输出层，一般使用 Softmax 函数获得同分布最高概率作为输出结果。此外，可以加入 Batch Normalization (BN) 层，让下一层的输入数据具有相同的分布。如果遇到神经网络训练时收敛速度慢，或梯度爆炸或者梯度消失等无法训练的状况都可以尝试加入 BN 层，然后观察其训练结果。

## 3.2 损失函数

在机器学习任务中，大部分监督学习算法都会有一个目标函数（Objective Function），算法对该目标函数进行优化，称为优化算法的过程。例如在分类或者回归任务中，使用损失函数（Loss Function）作为其目标函数对算法模型进行优化。

在第 2 章中，简单介绍了均方误差（MSE）损失函数和交叉熵损失函数，如图 3-7 所示，可以清晰地观察到不同的损失函数在梯度下降过程中的收敛速度和性能都是不同的。特别是针对自定义的深度神经网络，深度神经网络中的输出结果包括分类和回归问题，损失函数的模型会变得更加复杂。因此了解损失函数的类型并掌握损失函数的使用技巧，有助于加深对深度学习的认知。

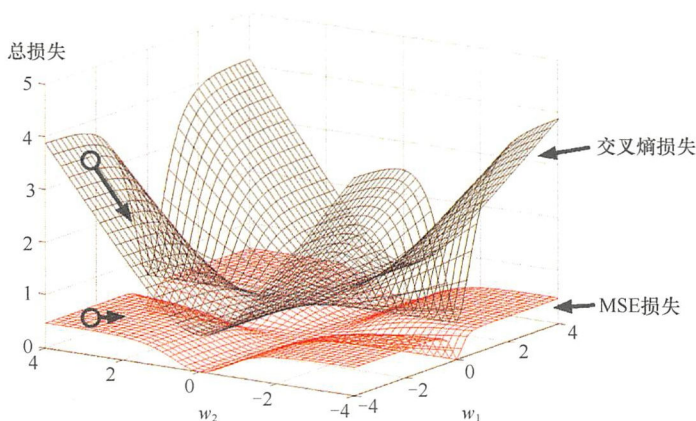


图 3-7 使用了 MSE 均方误差损失函数和交叉熵损失函数。MSE 损失函数收敛速度慢，可能会陷入局部最优解；而交叉熵损失函数的收敛速度较 MSE 快，且较为容易找到函数最优解

### 3.2.1 损失函数的定义

在神经网络中，损失函数用来评价网络模型输出的预测值  $\hat{Y} = f(X)$  与真实值  $Y$  之间的差异。这里使用  $L(Y, \hat{Y})$  来表示损失函数，它是一个非负实值函数。损失值越小，网络模型的性能就越好，所以优化算法目的就是让损失函数尽可能的小。

假设网络模型中有  $N$  个样本，样本的输入和输出向量为  $(X, Y) = (x_i, y_i), i \in [1, N]$ ，那么总损失函数  $L(Y, \hat{Y})$  为每一个输出预测值与真实值的误差之和：

$$L(Y, \hat{Y}) = \sum_{i=0}^N l(y_i, \hat{y}_i) \quad (3-6)$$

值得注意的是，机器学习问题主要分为回归和分类问题，对分类模型和回归模型进行评估时会使用不同的损失函数，下面将分别对回归模型和分类模型的损失函数进行介绍。

#### 分类与回归的区别

分类任务输入的是离散数据，目的是寻找决策边界，对输入的数据进行有效的分类，如预测明天是雨天还是晴天为一个分类任务。

回归任务输入的是连续数据，目的是找到最优的拟合方法，如预测明天的气温是多少摄氏度属于一个回归任务。

## 3.2.2 回归损失函数

### 1. 均方误差损失函数

运用均方误差 (Mean Squared Error Loss, MSE) 的典型回归算法有线性回归 (Linear Regression)。见【代码清单 3-6】，均方误差损失函数的定义如下：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2 \quad (3-7)$$

【代码清单 3-6】均方误差损失函数

```
def mean_squared_error(y_true, y_pred):
    ''' 均方误差损失函数实现 '''
    return np.mean(np.square(y_pred - y_true), axis=-1)
```

实际上，均方误差损失函数的计算公式可以看作欧式距离的计算公式。欧式距离的计算简单方便，而且是一种很好的相似性度量标准，因此通常使用 MSE 作为标准的衡量指标。

如果想要知道自己的神经网络模型损失函数是否选择正确，那么可以训练两个损失函数不同的网络模型，模型 A 损失函数使用均方误差，模型 B 则采用其他。训练后对比模型 A 与 B 的损失值曲线，如果 B 的损失值普遍比 A 小且收敛速度比 A 快，那么证明模型 B 的训练效果比一般的训练效果好，否则就应该选择其他损失函数。

另外由于均方误差损失函数对异常值非常敏感，平方操作会放大异常值，于是学者们又相继提出了平均绝对误差损失、均方误差对数损失、平均绝对百分比误差损失等损失函数来避免该问题。

### 2. 平均绝对误差损失函数

平均绝对误差损失 (Mean Absolute Error Loss, MAE)，是对数据的绝对误差求平均。见【代码清单 3-7】，平均绝对误差损失函数的定义如下：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{N} \sum_{i=1}^N |\hat{y}_i - y_i| \quad (3-8)$$

【代码清单 3-7】平均绝对误差损失函数

```
def mean_absolute_error(y_true, y_pred):
    ''' 平均绝对误差损失函数实现 '''
    return np.mean(np.abs(y_pred - y_true), axis=-1)
```

### 3. 均方误差对数损失函数

见【代码清单 3-8】，均方误差对数损失 (Mean Squared Log Error Loss, MSLE) 的定义如下：



$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{N} \sum_{i=1}^N (\log \hat{y}_i - \log y_i)^2 \quad (3-9)$$

#### 【代码清单 3-8】均方误差对数损失函数

```
def mean_squared_logarithmic_error(y_true, y_pred):
    ''' 均方误差对数损失实现 '''
    first_log = np.log(np.clip(y_pred, 10e-6, None) + 1.)
    second_log = np.log(np.clip(y_true, 10e-6, None) + 1.)
    return np.mean(np.square(first_log - second_log), axis=-1)
```

均方误差对数损失函数的实现代码中涉及对数计算，为了避免计算  $\log 0$  没有意义，因此加入一个很小的常数  $\varepsilon = 10^{-6}$  作为计算补偿。

#### 4. 平均绝对百分比误差损失函数

见【代码清单 3-9】，平均绝对百分比误差损失（Mean Absolute Percentage Error Loss, MAPE）的定义如下：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{N} \sum_{i=1}^N \frac{100 \times |\hat{y}_i - y_i|}{y_i} \quad (3-10)$$

#### 【代码清单 3-9】平均绝对百分比误差损失函数

```
def mean_absolute_percentage_error(y_true, y_pred):
    ''' 平均绝对百分比误差损失函数实现 '''
    diff = np.abs((y_pred - y_true) / np.clip(np.abs(y_true), 10e-6, None))
    return 100 * np.mean(diff, axis=-1)
```

#### 5. 回归损失函数总结

尽管针对回归模型的损失函数有很多种，但是均方误差仍然是使用最广泛的，并且在大部分情况下，均方误差有着不错的性能，因此被用作损失函数的基本衡量指标。MAE 则会比较有效地惩罚异常值，如果数据异常值较多，需要考虑使用平均绝对误差损失作为损失函数。一般情况下，为了不让数据出现太多异常值，可以对数据进行预处理操作。

均方误差对数损失与均方误差的计算过程类似，多了对每个输出数据进行对数计算，目的是缩小函数输出的范围值。平均绝对百分比误差损失则计算预测值与真实值的相对误差。均方误差对数损失与平均绝对百分比误差损失实际上是用来处理大范围数据（如  $[-10^5, 10^5]$ ）的，但是在神经网络中，我们常把输入数据归一化到一个合理范围的（如  $[-1, 1]$ ），然后再使用均方误差或者平均绝对误差损失来计算损失。

为什么神经网络中常把输入数据归一化到合理范围内？

因为小范围的数据方便 GPU/CPU 进行计算，并能提高浮点运算精度，方便观察数据的变化趋势和变化形式。

### 3.2.3 分类损失函数

#### 1. Logistic 损失函数

神经网络中涉及多分类问题时最常用的是 Logistic 损失函数。神经网络模型会为每一个分类产生一个有效的概率。为了让某一分类的概率最大，而引入了最大似然估计函数。

在最大似然估计函数中，定义一个损失函数为  $\text{loss}(\mathbf{Y}, P(\mathbf{Y}|\mathbf{X}))$ ，公式表示样本  $\mathbf{X}$  在分类  $\mathbf{Y}$  的情况下，使概率  $P(\mathbf{Y}|\mathbf{X})$  达到最大值。换句话说，就是利用已知的样本  $\mathbf{X}$  分布，找到最有可能的参数，使样本  $\mathbf{X}$  属于  $\mathbf{Y}$  的概率  $P(\mathbf{Y}|\mathbf{X})$  最大。假设二分类有  $P(Y=1|\mathbf{X})=Y$ 、 $P(Y=0|\mathbf{X})=1-Y$ ，因此对于多分类有：

$$P(\mathbf{Y}|\mathbf{X}) = y_i^{y_i} \times (1 - y_i)^{1-y_i} \quad (3-11)$$

在这里使用最大似然函数，目的是使每个分类都最大化，预测其所属正确分类的概率：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \prod_{i=0}^N \hat{y}_i^{y_i} \times (1 - \hat{y}_i)^{1-y_i} \quad (3-12)$$

#### 2. 负对数似然损失函数

为了方便数学运算，在处理概率乘积时通常把最大似然函数转化为概率的对数，这样可以把最大似然函数中的连乘转化为求和。在前面加一个负号之后，最大化概率  $P(\mathbf{Y}|\mathbf{X})$  等价于寻找最小化的损失。最后，Logistic 损失函数变成了常见的负对数似然函数（Negative Log Likelihood Loss）：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = -\sum_{i=0}^N y_i \times \log \hat{y}_i + (1 - y_i) \times \log(1 - \hat{y}_i) \quad (3-13)$$

#### 3. 交叉熵损失函数

见【代码清单 3-10】，由于 Logistic 损失函数和负对数似然损失函数都只能处理二分类问题，从两个类别扩展到  $M$  个类别，于是有了交叉熵损失函数（Cross Entropy Loss）：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = -\sum_{i=1}^N \sum_{j=1}^M y_{ij} \times \log \hat{y}_{ij} \quad (3-14)$$

#### 【代码清单 3-10】交叉熵损失函数

```
def cross_entropy(y_true, y_pred):
    ''' 交叉熵损失实现 '''
    return -np.sum(y_true * np.log(y_pred + 10e-6))
```

#### 4. Hinge 损失函数

运用 Hinge 损失的典型分类器是 SVM 算法，因为 Hinge 损失可以用来解决间隔

最大化问题。当分类模型需要硬分类结果的，例如分类结果是 0 或 1、-1 或 1 的二分类数据，Hinge 损失无疑是最方便的选择。见【代码清单 3-11】，Hinge 损失函数的定义如下：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{N} \sum_{i=1}^N \max(0, 1 - \hat{y}_i \times y_i) \quad (3-15)$$

#### 【代码清单 3-11】Hinge 损失函数

```
def hinge(y_true, y_pred):
    '''Hinge 损失实现'''
    return np.mean(np.maximum(1. - y_true * y_pred, 0.), axis=-1)
```

### 5. 指数损失函数

见【代码清单 3-12】，使用指数（Exponential）损失函数的典型分类器是 AdaBoost 算法：

$$\text{loss}(\mathbf{Y}, \hat{\mathbf{Y}}) = \sum_{i=1}^N e^{-y_i \times \hat{y}_i} \quad (3-16)$$

#### 【代码清单 3-12】指数损失函数

```
def exponential(y_true, y_pred):
    return np.sum (np.exp(-y_true * y_pred))
```

## 3.2.4 神经网络中常用的损失函数

神经网络中的损失函数可以自定义，前提是需要考虑数据本身和用于求解的优化方案。换句话说，自定义损失函数需要考虑输入的数据形式和对损失函数求导的算法。自定义损失函数其实是有些难度的，在实际工程项目上，结合激活函数来选择损失函数是常见的做法，常用组合有以下 3 种。

### 1. ReLU + MSE

均方误差损失函数无法处理梯度消失问题，而使用 Leak ReLU 激活函数能够减少计算时梯度消失的问题，因此在神经网络中如果需要使用均方误差损失函数，一般采用 Leak ReLU 等可以减少梯度消失的激活函数。另外，由于均方误差具有普遍性，一般作为衡量损失值的标准，因此使用均方误差作为损失函数表现既不会太好也不至于太差。

### 2. Sigmoid + Logistic

Sigmoid 函数会引起梯度消失问题：根据链式求导法，Sigmoid 函数求导后由多个 [0, 1] 范围的数进行连乘，如其导数形式为  $s(x)[1-s(x)]$ ，当其中一个数很小时，连成后会无限趋近于零直至最后消失。而类 Logistic 损失函数求导时，加上对数后

连乘操作转化为求和操作，在一定程度上避免了梯度消失，所以我们经常可以看到 Sigmoid 激活函数 + 交叉熵损失函数的组合。

### 3. Softmax + Logisitc

在数学上，Softmax 激活函数会返回输出类的互斥概率分布，也就是能把离散的输出转换为一个同分布互斥的概率，如 (0.2, 0.8)。另外，类 Logisitc 损失函数是基于概率的最大似然估计函数而来的，因此输出概率化能够更加方便优化算法进行求导和计算，所以我们经常可以看到输出层使用 Softmax 激活函数 + 交叉熵损失函数的组合。

## 3.3 超参数

在机器学习中，有两种类型参数：一种是与网络模型相关的参数，另一种是与网络模型调优训练相关的参数。与网络模型相关的参数有权重参数  $W$ 、偏置  $b$  等。与模型调优训练有关的参数目的是让模型训练的效果更好、收敛速度更快，而这些调优参数被称为超参数 (Hyper Parameters)。

超参数选择的目标是：保证神经网络模型在训练阶段既不会拟合失败，也不会过度拟合，同时让网络尽可能快地学习数据结构特征。下面对网络超参数的学习率 (Learning Rate) 和动量 (Momentum) 进行讲解。

### 3.3.1 学习率

梯度下降算法被广泛应用于最小化模型误差的参数优化算法，其公式如下：

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta} \quad (3-17)$$

其中  $\eta \in \mathbf{R}$  为学习率， $\theta$  为网络模型的参数， $L = L(\theta)$  为关于  $\theta$  损失函数， $\partial L(\theta) / \partial \theta$  为损失函数对参数  $\theta$  一阶导数 (梯度误差)。网络参数  $\theta$  的更新依赖于梯度误差与学习率：学习率越大，参数  $\theta$  的更新步长越大；学习率越小，参数  $\theta$  的更新步长越小。

在神经网络的训练阶段，调整梯度下降算法的学习率可以改变网络权重参数的更新幅度。当大的损失和陡峭的梯度与学习率相结合时，下一步长会很大；当误差很小且梯度比较平坦时与学习率相结合时，下一步长会缩短。

为了使梯度下降法具有更好的性能，我们需要把学习率的值设定在合适的范围内，因为学习率决定了参数能否移动到最优值和参数移动到最优值的速度。如果学习率过大 (例如  $\eta = 1$ ，最优值为 0.3，那么最后的误差值可能在  $-0.7 \sim 1.3$  之间来回



跳动)，权重参数很可能会越过最优值，最后在误差最小的一侧来回跳动，永不停止。反之，如果学习率过小（例如 $\eta=1e^{-5}$ ），网络可能需要很长的优化时间，优化的效率过低，最终导致算法长时间无法收敛（如图 3-8 所示）。

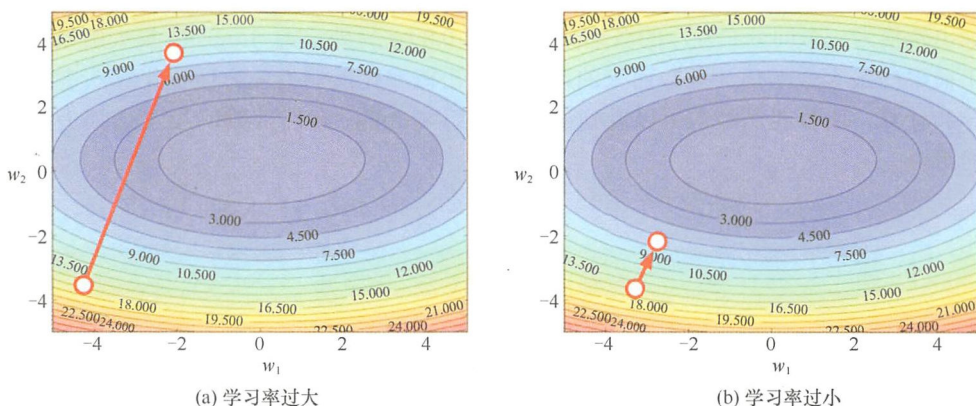


图 3-8 学习率过大，越过最优值，最后在最优值间来回波动；  
学习率过小，训练时间长（图中坐标原点为最优值）

学习率的设置对于算法性能的表现至关重要，一个好的学习率，对应的误差曲线应该具有很好的平滑性，并且最终使得网络达到最优的性能。一般情况下，高学习率使得误差下降得很快，低学习率可以使得误差下降平滑。

设定学习率时，我们可以选择一个合适的值（如 0.1），然后多次尝试不同的学习率，检查在最开始时网络误差梯度下降的趋势，及其在网络训练初期获得最佳的速度和准确性，最后在微调网络时再次调整网络的学习率。另外，我们可以让学习率随迭代次数分阶段衰减。在网络训练初期使用高学习率，当误差降低幅度减少时转而采用较低的学习率，让误差继续平滑下降，这样就可以让模型训练得到更好的效果。例如将学习率的初始值设为 0.1，若在验证集上误差性能不再提高，可以将学习率除以 2 或者 5，如此循环，直到算法收敛。

### 3.3.2 动量

动量（Momentum）的物理意义是：当我们把球推下山时，球会不断地累积其动量，速度会越来越快（直到其最大速度），当球遇到上坡时其动量就会减少。

参数更新时也可以模仿物理中的动量：当梯度保持相同方向维度时，动量不断增大，梯度方向在不停变化的维度上，动量持续减少，因此可以加快收敛速度并减少震荡。网络中的参数通过动量来更新，参数向量会在任何有持续梯度的方向上增

加速度。其公式为：

$$\theta \leftarrow \mu * \theta - \eta \frac{\partial L}{\partial \theta} \quad (3-18)$$

其中， $\mu \in \mathbf{R}$  为动量，动量系数取值为  $0 \sim 1$ 。该式表明当前梯度方向与前一步的梯度方向一样，那么就增加这一步的权值更新，否则减少参数更新。最终达到在一定程度上增加稳定性，加快学习速率，并且有一定的摆脱局部最优的能力，如图 3-9 所示。

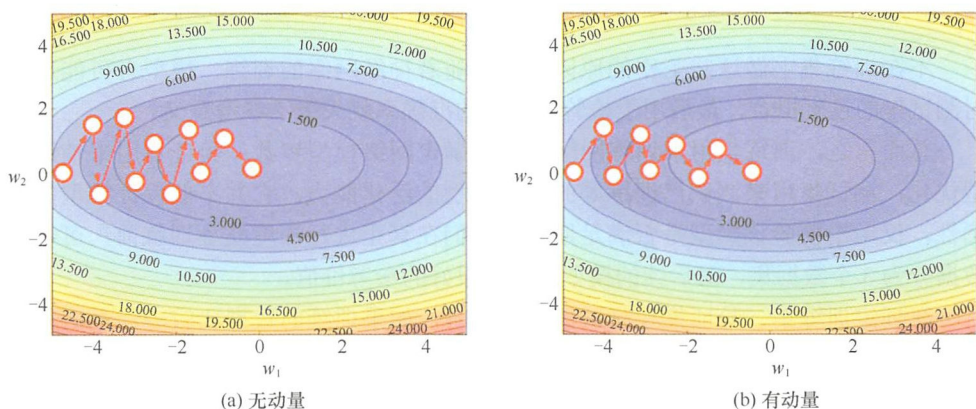


图 3-9 动量示例图。图中更新参数的方法均为小批量随机梯度下降算法，其中（a）为只有学习率没有动量的情况下，梯度持续波动最终走向收敛；（b）为带有动量和学习率，梯度波动幅度减少并且加快了收敛速度

对于动量的设置，常用的取值为 0.5、0.9、0.95 或者 0.99。在网络训练的起始阶段，由于梯度可能会很大，所以初始值一般设定为 0.5；当梯度下降到一定程度时，可以改为 0.9 或者更大。

## 3.4 深度学习的技巧

### 3.4.1 数据集准备

如图 3-10 所示，我们常常把原始数据集分为 3 部分：训练集（training data）、验证集（validation data）和测试集（testing data）。训练集就是用来训练的数据集合，测试集就是训练完后用来测试训练后模型的集合，那么验证集有什么用呢？在模型训练过程中，可以通过验证集来观察模型的拟合情况，如果出现过度拟合，则及时

停止训练；还可以通过验证集来确定一些超参数（如根据验证集的精确率来确定迭代次数，根据验证集的收敛情况确定学习率等）。

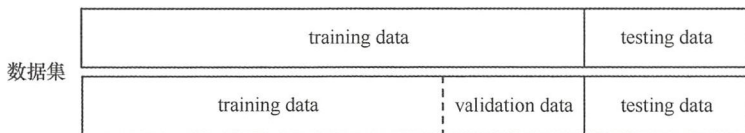


图 3-10 数据集准备：上为训练集和测试集；下为训练集、验证集和测试集

不在训练集上做这些工作的原因在于，随着神经网络的训练，模型有可能造成对测试集的过度拟合，最后用于检验测试集的准确率就失去了参考意义。

总而言之，训练集用来训练网络模型或确定网络模型参数；验证集用来辅助模型优化；测试集用来测试已训练好的网络模型的泛化能力，它并不能保证模型的正确性，仅用于检验该模型是否满足期望。我们在真正进行深度模型训练前，可以把数据分为 3 个数据集，对后续的模型训练有很好的参考价值。

在划定训练集、验证集和测试集的比例时，如果训练的数据比较少，最后得到的估计参数会有较大的偏差；如果测试的数据少，最后统计值会有较大的偏差。针对该问题，我们需要对数据集中的数据进行划分，让不同数据集之间各自的方差不会相差太大。

假设现在有 100 条数据，这时无论怎么交叉数据集或者通过哪种方式去划分，最后的效果都不会太理想。如果有 1000 万条数据，那么选择 8:2 还是 9:1 可能差别不会太大，所以关键是希望拥有更多的数据。下面的一些方法可以帮助我们更好地选择集合样本数：

- (1) 将数据分为训练集和测试集，比例可为 8:2；
- (2) 将 (1) 中训练集中的数据继续分为训练集和验证集两部分，最终比例为 8:2:2.5；
- (3) 对训练集随机抽样为小集合，用来训练，并在验证集上记录性能；
- (4) 打散训练集、测试集、验证集中数据的排列方式，并重新训练；

(5) 从训练集中随机抽取 80% 的数据进行多次训练，把训练集中剩下的数据作为验证集，然后观察不同抽取结果在测试集中的性能，把最好的一次训练结果当作最终模型。

### 3.4.2 数据集扩展

在深度学习中，我们需要大量已经标注好的样本用于训练，可是收集海量的数据并不是一件容易的事情。假设应用场景是对图片进行分类，则需要已经标注的图



片，收集图片数据的常用方法是对门户网站、搜索引擎等进行定向爬虫，根据页面标注对图片进行分类。无论使用什么方法去收集数据，都无法避免消耗大量的人力资源。尤其是在医学图像中，每一张正样本图片背后都有一位正在承受病痛折磨的病人，像医学图像这些稀缺的正样本更加难以收集。

在图像数据中，对数据集进行扩展的常用方法包括：对图片进行角度偏移、左右偏移、上下偏移、随机放大或者缩小、水平翻转等。如果将上述方法组合排列起来，例如对图片进行放大后再进行角度偏移，这样将会产生大量的图片（如图 3-11 所示）。除了简单地对图像进行几何上的形变，我们还可以对图片的饱和度、亮度、色彩进行小范围的幂次缩放或者乘法缩放等数学操作。另外还可以在不破坏图像质量的前提下，对图像的每个像素进行整体加减法等操作。



图 3-11 左上角第一张图为原图，余下的图片均为通过旋转、扭曲、拉伸、增加噪声等图像操作形成，目的是为了增加大量的数据用于网络训练学习其特征

【代码清单 3-13】对输入的图片进行数据扩展的 Python 代码，使用 Keras 自带的 ImageDataGenerator 对象产生更多的图片数据，rotation\_range 是控制旋转的角度，shift\_range 是左右上下的偏移度，shear\_range 是进行剪切变换的程度，zoom\_range 是放大缩小的倍数。

#### 【代码清单 3-13】Keras 数据预处理

```
from keras.preprocessing.image import ImageDataGenerator

data_generate = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```



接着对 ImageDataGenerator 对象使用 flow 方法，将该图像生成器实例化。如【代码清单 3-14】所示，由于变换角度多样，并且组合方式非常多，因此需要设定产生图片的数量 i，当迭代次数达到 20 次后，则通过 break 语句跳出循环。

【代码清单 3-14】Keras 数据预处理（续）

```
from keras.preprocessing.image import img_to_array, load_img

img = load_img('cat.jpg')
x = img_to_array(img) # (3, 150, 150)
x = x.reshape((1,) + x.shape) # (1, 3, 150, 150)

i = 0
for batch in data_generate.flow(x, batch_size=1, save_prefix='cat', save_format='jpeg'):
    i += 1
    if i > 20:
        break
```

最终，代码的运行效果如图 3-11 所示，左上角第一张图片为原始的猫图，其余为通过组合角度偏移、左右上下偏移、随机放大缩小产生新的猫图。部分研究者的数据表明，使用了该数据扩展方法之后可以在 ImageNet 分类模型中使预测准确率提升 2% ~ 5%，并且有效地减少了过度拟合。

### 3.4.3 数据预处理

假设在数据矩阵中通过不同的提取方式获得不同列的数据，有些数据值特别大、有些数据则是小数，例如 np.array([1024, 222, 0.0216, 0.0412, 19566])，这时由于数据高度不对称，算法无法处理所有不在同一维度的数据，最终可能会导致网络模型训练失败。因此在获得海量数据集后，有必要对数据进行预处理操作，使得数据分布无偏低方差，如图 3-12 所示。

下面来介绍 4 种预处理的常用方法：

- 0 均值（Zero Centralization）
- 归一化（Normalization）
- 主成分分析（Principal Component Analysis, PCA）
- 白化（Whitening）

#### 1. 0 均值

0 均值是最常用的数据预处理方法，通过用数据中的每一维度的数据值减去所在维度的数据均值，最后呈现的数据就是 0 均值数据。如【代码清单 3-15】所示，在 numpy 中，可以用 np.mean 方法实现，如图 3-13 所示。

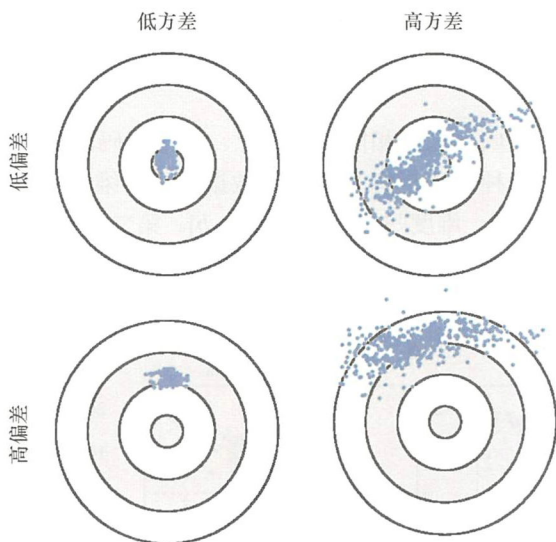


图 3-12 数据分布示例图。通常来说，我们希望数据是无偏低方差的，如左上角所示；  
 右上角数据为无偏高方差，左下角为高偏差低方差，右下角为高偏差高方差

### 【代码清单 3-15】0 均值实现

```
>>> X = np.random.rand(2,5) # 随机产生一个 2x5 的数组
array([[ 0.05885589,  0.03326574,  0.72343443,  0.97790419,  0.48032038],
       [ 0.14338263,  0.3300891 ,  0.05866592,  0.9163435 ,  0.99844258]])

>>> M = np.mean(X, axis = 0) # 对数组 x 求每一维度的平均值
array([ 0.10111926,  0.18167742,  0.39105018,  0.94712385,  0.73938148])

>>> X -= M # 0 均值操作
array([[ -0.04226337, -0.14841168,  0.33238426,  0.03078035, -0.2590611 ],
       [  0.04226337,  0.14841168, -0.33238426, -0.03078035,  0.2590611 ]])

>>> X -= np.mean(X, axis = 0) # 0 均值操作可以直接写成
```

在深度学习中，数据输入神经网络之前常用 0 均值方法进行处理。对于图像来说，可以更简单地用所有像素值减去同一个均值，也可以对彩色图像 RGB 的 3 个通道分别进行 0 均值操作。如【代码清单 3-16】所示，减去的不是图像的均值 125，而是 ImageNet 所有图像在其所在通道的均值。

### 【代码清单 3-16】图像 0 均值操作

```
>>> X = load_img("cat.jpg") # 读取图片数据
>>> X[:, :, 0] -= 103-939 # 减去第一个通道中训练集的均值
```

```
>>> X[:, :, 1] -= 116.779      # 减去第二个通道中训练集的均值
>>> X[:, :, 2] -= 123-68      # 减去第三个通道中训练集的均值
```

## 2. 归一化

归一化是指将数据归一化到相同的尺度上。如【代码清单 3-17】所示，归一化有两种常用的方法：一种方法是将 0 均值后数据的每一维除以每一维的标准差；另一种方法是把数据中的每一维度归一化到区间  $[a, b]$ 。第二种归一化方法只适用于数据的不同维度应该具有相同的重要性时，也就是每一维度数据的权重一样时，如图 3-13 所示。

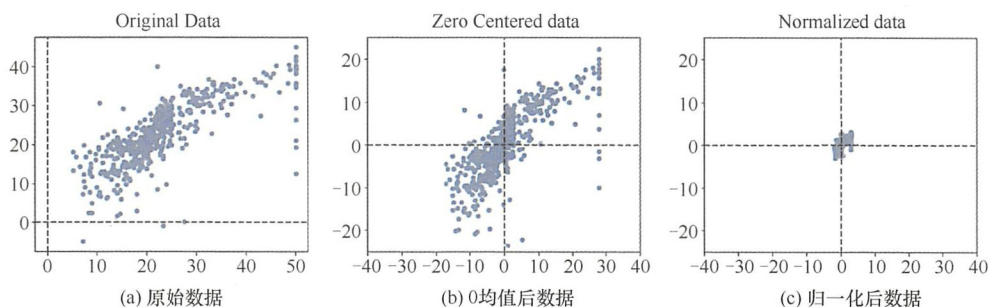


图 3-13 (a) 中的数据来自 sklearn 的 boston 数据集，大部分数据都分布在 0 以上；

(b) 为经过 0 均值处理后的数据，数据以坐标原点 (0,0) 为中心分布；

(c) 为归一化处理后的数据，每一维度的数据根据标准差进行了缩放

### 【代码清单 3-17】数据归一化实现

```
>>> X -= np.mean(X, axis = 0)
>>> X /= np.std(X, axis = 0)      # 0 均值后每一维除以每一维的标准差
>>> x_normed = x / x.max(axis=0)  # 归一化为 (0, 1) 之间
```

例如数组 `np.array([1024, 222, 0.0216, 0.0412, 19566])`，当输入的数据大小参差不齐时，有必要对数据进行归一化操作之后，再乘以该维度数据的权重值。对于图像来说，RGB 图像的像素区间一般固定在 0 ~ 255 之间，因此不需要对图像进行归一化处理。

## 3. 主成分分析 PCA

主成分分析 (Principal Component Analysis, PCA) 是一种常用的数据分析方法，其目的是用来寻找有效表示数据主轴的方向 (如图 3-14 所示)。

主成分分析通过线性变换将原始数据变换为一组各维度线性无关的数据，可用于提取数据的主要特征分量，常用于高维数据的降维。在机器学习或者数据挖掘算法中，数据预处理时可以通过主成分分析进行降维，有效减少后续计算量、降低数据噪声，使得神经网络、SVM 等分类器呈现更好的效果。

更正式地说，主成分分析把原始数据的  $n$  个特征用  $m$  ( $m < n$ ) 个特征取代，新特征是旧特征的线性组合。这些线性组合最大化样本方差，尽量使新的  $m$  个特征互不相关，从旧特征到新特征的映射捕获数据中的固有变异性。

协方差矩阵 (Covariance Matrix) :

$$\text{COV}(X, Y) = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{n-1} \quad (3-19)$$

协方差矩阵对角线上的元素表示数据的方差，假设协方差矩阵为  $M$ ，那么  $M_{ij}$  ( $i \neq j$ ) 表示第  $i$  维和第  $j$  维数据的特征相关性。

奇异值分解 (Singular Value Decomposition, SVD) :

$$A = USV^T \quad (3-20)$$

矩阵奇异值分解可以将一个复杂的矩阵用更小更简单的 3 个子矩阵的相乘来表示，这些小矩阵描述矩阵的重要特性。对矩阵进行奇异值分解，得到 3 个矩阵 ( $U$  酉矩阵,  $S$  对角矩阵,  $V^T$  共轭转置矩阵)。

假设  $A$  是一个  $N \times M$  的矩阵，那么得到的酉矩阵  $U$  是一个  $N \times N$  的矩阵 ( $U$  中的向量称为左奇异向量，列为特征向量)，对角矩阵  $S$  是一个  $N \times M$  的矩阵 (除对角线外其余元素为 0，对角线上元素称为奇异值)，共轭转置矩阵  $V^T$  是一个  $M \times M$  的矩阵 ( $V$  中的向量称为右奇异向量)。

为了对数据去相关性，需要把原始的数据投影到特征向量上。SVD 分解后得到的  $S$  矩阵的列是相互正交的向量，因此它们可以被看成基向量，用于去数据相关性。这种投影相当于将数据旋转并投影到新的基向量轴上。

在对数据进行主成分分析之前，不能直接求数据的协方差矩阵，首先需要把数据经过 0 均值处理，然后才能计算其协方差矩阵，得到数据不同维度之间的相关性。接着可以对协方差矩阵进行 SVD 分解，得到 3 个矩阵 ( $U$  酉矩阵,  $S$  对角矩阵,  $V$  共轭转置矩阵)。

SVD 分解一个很好的特性，因为返回的  $U$  矩阵是按照其特征值的大小排序的，排在前面的就是主方向，因此可以通过选取前几个特征向量来降低数据的维度，这里就是主成分分析的降维方法。

经过上述操作后，就能将原始数据从  $n = N \times D$  维转变成  $m = N \times 100$  维， $N \times 100$  的矩阵中保留了原数据的最大方差。

#### 【代码清单 3-18】主成分分析实现

```
>>> X -= np.mean(X, axis = 0)           # 对输入的数据 X 进行 0 均值计算
>>> COV= np.dot(X.T, X) / X.shape[0]   # 获取数据矩阵 X 的协方差矩阵
>>> U, S, V = np.linalg.svd(COV)       # 协方差矩阵 SVD 分解
```



```
>>> Xrot = np.dot(X, U)          # 数据去相关性
>>> PCA = np.dot(X, U[:, :100])  # 获得去相关性的前 100 列数据
```

#### 4. 白化

白化 (Whitening) 的目的是降低数据的冗余性。我们希望通过白化操作使得数据具有如下性质：

- 特征之间相关性较低；
- 所有特征具有相同的方差 (如图 3-14 所示)。

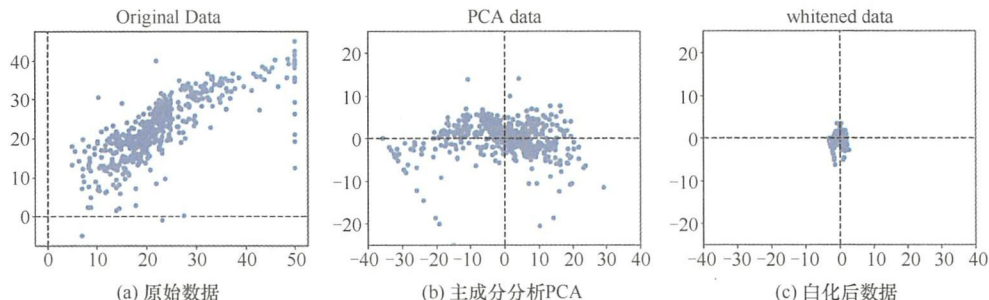


图 3-14 (a) 中的数据来自于 sklearn 的 boston 数据集, 大部分数据都分布在 0 以上; (b) 为 PCA 处理后的数据集, 数据以坐标原点为中心, 并且根据协方差矩阵的特征基进行了旋转; (c) 为白化处理的数据, 每一维的数据根据特征值进行了缩放, 白化后的数据符合高斯分布

在前面的主成分分析算法中, 在对数据进行降维后, 已经消除了原始数据之间的相关性。因此白化的过程只需要把通过主成分分析去相关后的数据, 从对角矩阵再变成单位矩阵, 使得数据具有相同的方差。

【代码清单 3-19】中对对角矩阵  $S$  中加入了一个很小的常量 ( $1e^{-6}$ ), 是为了防止主成分分析的矩阵元素除以 0。白化的缺点在于扩大数据的噪声, 因为它将数据的维度拉伸到相同的大小, 把对角矩阵变成单位矩阵。

#### 【代码清单 3-19】数据白化实现

```
Xwhiten = PCA / np.sqrt(S + 1e-6)  # 除以特征值, 也就是奇异值的平方根
```

需要注意的是, 在深度学习的图像处理中, 甚少用到主成分分析和白化操作。因为主成分分析和白化是为了对数据进行完整性操作, 图像中的像素值已经包含了丰富且相关联的完整信息。但是, 对图像数据进行 0 均值是非常重要的, 因此在任何一个深度学习的例子中输入数据都会进行 0 均值操作。而归一化也是很常见的数据预处理操作, 其负责数据的一致性。

另外值得注意的一点是, 防止把对数据预处理操作统一应用到训练集、验证集

和测试集中。也就是先对 3 个数据集中的全部数据进行统一预处理，然后分别进行训练、验证、测试，这样的顺序是错误的。以 0 均值为例，对输入数据的操作正确方式为：首先对训练集的数据求均值，然后分别在验证和测试时减去训练集的均值，而不是减去所有或者自身的均值。

### 3.4.4 网络的初始化

定义完深度学习的神经网络模型之后，需要对网络的权重  $W$  和偏置  $b$  进行初始化。因为基本上大部分深度学习框架已经帮我们处理好了网络初始化这一步骤，可能部分读者会觉得把所有参数初始化为 0 并没有什么问题。现在假设，所有权重的初始化为 0，那么网络中的每个神经元的输出都将是一样的，利用反向传播 BP 算法计算的梯度都一样，所有的参数更新也都一样，最后整个模型就失去了意义。

因此把模型的权重和偏置参数初始化为 0 是不科学的。为了让初始化的权重参数尽可能小，但是又不能为 0，这里可以参考高斯分布函数，使用独立高斯随机变量来选择初始化模型的权重和偏置（如图 3-15 所示）。

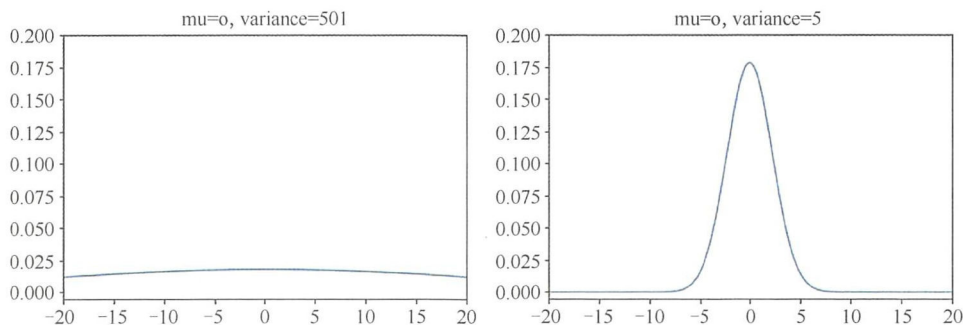


图 3-15 使用高斯函数生成的正态分布曲线模型。

在【代码清单 3-20】中，函数 `nerous()` 为单个神经元函数，假设输入层有 1000 个神经元，隐藏层有 1 个神经元，输入数据  $x$  为一个全 1 的向量，采用高斯分布来初始化权重矩阵  $w$ ，偏置  $b$  为 0。那么可以通过 `random` 函数产生一个满足正态分布的一维随机化向量，然后根据神经元的加权输出  $z = \sum_i w_i x_i + b$  公式输出。

接着对 `nerous` 函数迭代 100000 次，以期获得不同随机生成的权重参数所得到的加权输出，使用 `plt.hist` 显示其直方图结果（如图 3-16 所示）。

#### 【代码清单 3-20】使用随机高斯分布产生初始化网络参数

```
def nerous(sqrt = False, num = 1000, b = 0):
    ''' 神经元模型 '''
```

## 82 | 第3章 深度学习基础及技巧

```

x = np.ones(num)
if not sqrt:
    w = np.random.rand(num)
else:
    w = np.random.rand(num)/np.sqrt(num)
return np.dot(w, x) + b

# 迭代 100000 次获得不同随机生成的权重参数得到的加权输出
for i in range(100000):
    z1.append(neurous())

# 显示 100000 次加权输出的直方图
n, bins, patches = plt.hist(z1, 100, alpha=0.8)
plt.show()

```

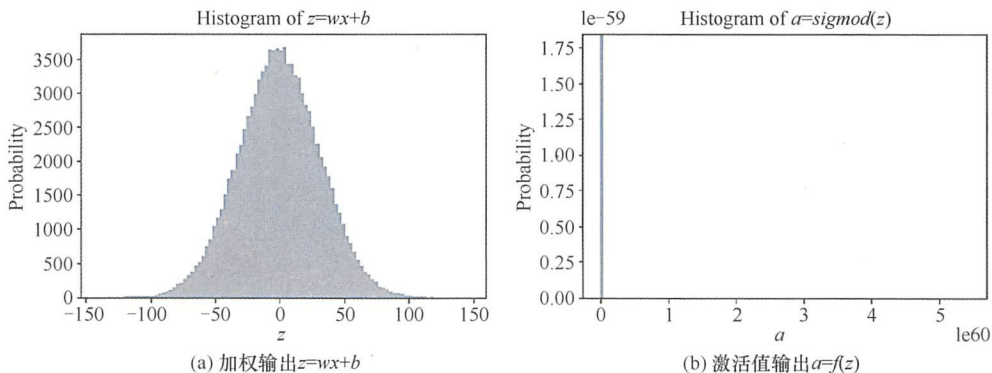


图 3-16 权重参数初始化使用高斯分布生成随机值。记录下迭代 100000 次随机生成初始化值情况，其中，(a) 为神经元的加权输出，(b) 为神经元的激活值输出

### 【代码清单 3-21】输出激活值

```

sigmoid = lambda x:1./1.+np.exp(-x)
a1 = sigmoid(np.array(z1))

```

从图 3-16 (a) 中可以看出，加权输出值  $z$  的取值范围较大，通过激活函数输出后得到的激活值基本上为 0 或者 1 两极，这就使得隐层神经元基本处于饱和状态。经过随机梯度下降算法对网络参数进行调整，仅仅会给隐层神经元的激活值带来极小的变化，这意味着输出神经元参数趋于饱和，也就是下一层的神经元输入饱和，后面通过反向传播 BP 算法对权重参数进行微调所带来的变化也不会太明显。最终网络学习速率非常慢，不仅让网络训练慢，而且很容易引起过度拟合。

产生上面的结果是因为随机产生权重参数中的标准差太大，下面来看让随机产生的权重参数归一化为均值 0，标准差为  $1/\sqrt{n}$  的高斯随机分布的效果。

如【代码清单 3-22】所示，对随机初始化的权重参数除以权重参数的开方 ( $w/\sqrt{n}$ )，以控制其标准差。最终使得输出的激活值能够在 0 和 1 之间分布，有效地解决了神经元的过饱和问题，既保证了网络模型中的所有权重参数在初始阶段都符合高斯分布的输入，也提高了算法的收敛速度（如图 3-17 所示）。

【代码清单 3-22】使用随机高斯分布产生初始化网络参数

```
z1 = []
for i in range(100000):
    # w = np.random.rand(num)/np.sqrt(num) # 新的权重设置
    z1.append(neurous(sqrt=True))
```

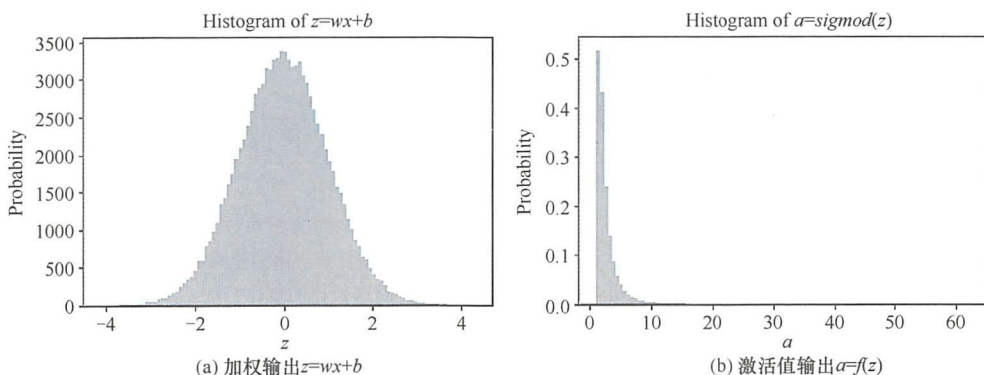


图 3-17 权重参数初始化使用高斯分布生成随机值。记录下迭代 100000 次随机生成初始化值情况，其中，(a) 为神经元的加权输出，(b) 为神经元的激活值输出

可是使用高斯随机变量产生的初始化参数仍然不够好，我们希望在初始化时就可以使训练的效果尽可能好，并且能够有效地缩短训练时间，减少过度拟合。

在实际的工程实践当中，为了避免重新训练网络模型参数花费大量的时间，可以使用 ImageNet 数据集已经预先训练好的模型，加载到网络中然后开始训练。因为预先训练好的模型参数已经根据某数据规则经过长时间的训练，网络参数经过一定程度的优化，直接使用该优化参数可以降低重新训练可能造成的优化失败，并减少过度拟合的情况，有效节省大量的时间，如【代码清单 3-23】所示。

【代码清单 3-23】Keras 加载预训练模型进行训练

```
model.load_weight("xxxx_pre_train.h5")

model.compile(loss="categorical_crossentropy",
              optimizer='adam',
              metrics=['accuracy'])
```



```
model.fit(X_train, Y_train, batch_size = 32, nb_epoch=10, verbose=1)
```

### 3.4.5 网络过度拟合

当一个模型从样本中学习到的特征不能够推广到其他新数据时，直接使用该模型对新数据进行预测很有可能出现过度拟合（Over Fitting）的情况（如图 3-18 所示）。过度拟合发生后，模型试图使用不相关的特征对新数据进行预测，最终导致网络模型预测的结果出错。

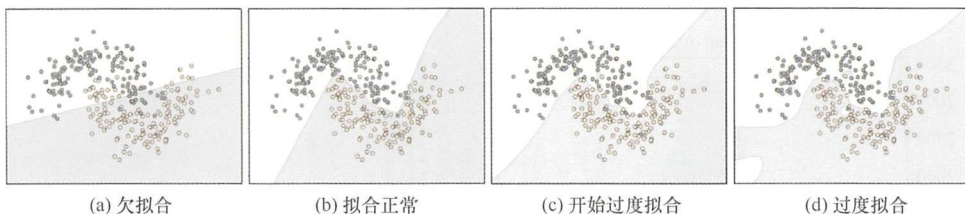


图 3-18 数据过度拟合示例。(a) 中模型过于简单，无法正确表达数能够正常对数据进行拟合；(b) 中模型适中，能够正常对数据进行拟合；从 (c) 开始，模型开始越来越复杂，虽然能够更好地表达训练数据集，但是由于过度拟合数据集中的噪声而忽略了数据的整体特征

下面来看如何判断网络模型是否过度拟合。假设在训练数据集上能够对数据进行很好的拟合，但是在训练数据集外的数据集上却不能够很好的拟合，从而获得更小的损失值，此时可以认为这个模型出现了过拟合的现象，出现该现象的主要原因是训练数据中存在噪音或者训练数据太少。

出现过度拟合时，一个简单直接有效的方法是增加训练集的数据，同时增加数据的多样性。但这是不够的，因为提升过的数据仍然可能是高度相关的。为了避免过度拟合，应该主要关注网络模型的“熵容量”——网络模型允许存储的信息量。如果网络能够存储更多信息，那么意味着能够存储和利用更多的特征，获得更好的性能，但是同时也会带来存储不相关特征的风险。此外，只能存储少量信息的网络模型，其存储的特征将会集中在真正相关的特征上，这样神经网络就拥有了更好的泛化性能。

现在已经有很多不同的方法来调整网络模型的“熵容量”，常见的选择是调整模型的参数数目，即模型的层数和每层的规模；另一种方法则是在网络权重更新时进行正则化约束。

#### 什么是泛化（Generalization）

泛化是指容错能力，有一定泛化能力就表明可以接受一定的错误输入，经过内部纠正后输出正确的结果。

### 3.4.6 正则化方法

正则化的最大作用是防止过度拟合，提高网络模型的泛化能力，具体实现方法是在损失函数中增加惩罚因子。

在机器学习中，无论是分类还是回归问题，都可能存在由于特征过多而导致的过度拟合问题，可以通过下面两种方法来解决：

- 减少特征，留下重要的、具有普遍性的特征；
- 惩罚不重要特征。

之所以需要进行深度学习，是希望网络能够捕捉到更多数据的高维特征，所以我们不希望减少网络所产生的高维数据特征，还可以通过惩罚不重要的特征防止过度拟合。惩罚不重要的特征使用的是正则化技术，包括 L1 范式、L2 范式、最大约束范式和引入 Dropout 层。

#### 1. L2 正则化

L2 正则化就是在损失函数后面增加上 L2 正则化项  $\frac{\lambda}{2n} \sum_i^n w_i^2$ 。L2 正则化公式为：

$$L = L_0 + \frac{\lambda}{2n} \sum_i^n w_i^2 \quad (3-21)$$

其中  $L_0$  为原始损失函数， $\frac{\lambda}{2n} \sum_i^n w_i^2$  为 L2 正则化项。L2 正则化项为所有权值的平方和  $(\sum_i^n w_i^2)$  除以训练集中的样本大小  $n$ ， $\lambda \in \mathbf{R}$  是引入的正则项系数，用来调节正则项和原始损失值  $L_0$  的比重，系数 1/2 是方便求导时进行约简。

对 L2 正则化公式进行求导后得到：

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L_0}{\partial \mathbf{w}} + \frac{\lambda}{n} \mathbf{w}$$

将上述公式代入梯度下降公式，L2 正则化后权值  $\mathbf{w}$  的更新为：

$$\mathbf{w} \leftarrow \left(1 - \eta \frac{\lambda}{n}\right) \mathbf{w} - \eta \frac{\partial L_0}{\partial \mathbf{w}} \quad (3-22)$$

没有使用 L2 正则化时权值  $\mathbf{w}$  前面的系数是 1，使用 L2 正则化后权值  $\mathbf{w}$  前面的系数为  $1 - \eta \frac{\lambda}{n}$ ，其中  $\eta$ 、 $\lambda$ 、 $n$  为正数，使得权值  $\mathbf{w}$  的系数恒小于 1。因此可以看出 L2 正则化就是用来惩罚特征的权值  $\mathbf{w}$  的，学术上称为权值衰减（Weight Decay）。

L2 正则化确实能够让权值变得更小，它可以用于防止过度拟合的原因在于更小的权值表示神经网络的复杂度更低、网络参数越小，这说明网络模型相对简单，越简单的模型引起过度拟合的可能性越小。

【代码清单 3-24】为在 Keras 的卷积层中加入 L2 正则化。L2 正则化是通过惩罚网络中的权重参数来实现的，因此我们可以指定哪一层网络中需要进行正则化惩罚权重参数，正则化原则是按照每一层来执行的。

【代码清单 3-24】在 conv2D 层中加入 L2 正则化

```
from keras.regularizers import l2

C1 = Conv2D(20, 4, 4, border_mode='valid', activation='relu', W_regularizer=l2(0.1))
```

## 2. L1 正则化

L1 正则化是在原始的损失函数后面加上一个 L1 正则化项  $\frac{\lambda}{n} \sum_i^n |w_i|$ ，即权值  $w$  绝对值的和，乘以  $\lambda/n$ 。L1 正则化公式为：

$$L = L_0 + \frac{\lambda}{n} \sum_i^n |w_i| \quad (3-23)$$

当权值为正时，更新后的权值变小；当权值为负时，更新后的权值变大。因此 L1 正则化的目的就是让权值趋向于 0，使得神经网络中的权值尽可能小，也就相当于减小了网络复杂度，防止过拟合（其数学推导跟 L2 正则化类似，这里不重新推导，有兴趣的读者可自行证明）。

在实际应用中，一般使用 L2 正则化。因为 L1 范式会产生稀疏解，具有一定特征选择能力，对求解高维特征空间比较有用；L2 范式主要是为了防止过度拟合。

## 3. 最大约束范式

最大约束范式比 L1、L2 正则化容易理解，就是约束权值（限制权值大小），对每个神经元的权重绝对值给予限制。实际操作中先对所有参数进行正常更新，然后通过限制每个神经元的权重矢量使其满足关系式：

$$\|\bar{w}\|_2 < c \quad (3-24)$$

其中， $c \in \mathbf{R}$  常用取值为 3 或者 4。最大约束范式的特点是对权值的更新进行了约束，即使学习率很大，也不会因网络参数发生膨胀导致过度拟合。

输入神经网络中的训练集数据越多，正则化系数的作用就越少。因为训练数据越多，出现过度拟合的可能性就会降低，需要正则化系数去纠正的需求就会降低。（这里所指的增加神经网络的训练集数据不是单单增加数据量，还包括增加数据的多样性。）

## 4. Dropout 层

L1、L2 正则化是通过修改损失函数实现的，而 Dropout 层则是通过修改神经网络的模型实现的。Dropout 层少了数学公式和推导，理解起来会更加容易，是训练深度网络常用的技巧。

图 3-19（a）所示为普通神经网络模型，其神经元之间使用全连接的方式。Dropout

层则是在神经网络训练时随机地让部分隐层神经元失效，进而不能对其权重参数和偏置进行更新，如图 3-19 (b) 所示。

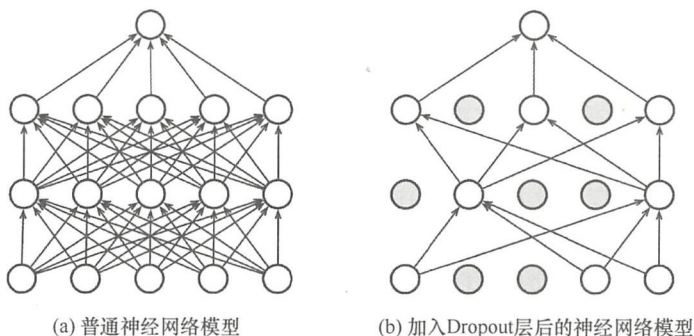


图 3-19 全连接的神经网络模型与带有 Dropout 的神经网络模型对比图

为什么 Dropout 层随机地让部分隐层神经元停止更新，会有助于防止过度拟合？学者 Hinton 的解释是：过度拟合可以通过阻止某些特征的协同作用来缓解。因为在每次训练时，神经元之间随机地被移除，可以让一个神经元的出现不依赖于另一个神经元，这样能够阻止特征互相依赖、减少错误信息的传递，防止过度拟合。

在实际操作中，Dropout 的概率通常为 50%，不过在验证集上这一数值是可以调整的。【代码清单 3-25】是 Dropout 的 Python 实现，其中  $x$  是本层网络的激活值， $level$  是 dropout 层中每个神经元被抹去的概率。

#### 【代码清单 3-25】神经网络 dropout 层实现

```
def dropout(x, level):
    # level 是概率值，必须在 0~1 之间
    if level < 0. or level >= 1:
        raise Exception('Dropout level must be in [0, 1].')

    retain_prob = 1. - level
    sample=np.random.binomial(n=1,p=retain_prob,size=x.shape)
    x *=sample           # 0、1 与 x 相乘屏蔽部分神经元
    x /= retain_prob
```

Dropout 层可以用于防止过度拟合，但不能通过都加入 Dropout 层地方式一劳永逸地解决问题。经过实际工程试验，当网络模型规模很小时，加入 Dropout 层的预测结果反而变差，在大型神经网络架构中加入 Dropout 层会增加网络的训练时间。实际上，对于小型的神经网络可以不用太过于担心过度拟合的问题，对于大型网络是否添加 Dropout 层要根据实际的效果进行对比。如果没有 Dropout 层验证集的误差和准确率与训练集相差无几，那么表明数据没有出现过度拟合，不加 Dropout 层反而可以



提高训练速度。

## 3.4.7 GPU的使用

### 1. 使用多 GPU

研究表明,同时使用 4 个 GPU 对一个网络模型进行训练,可以得到 2.5 ~ 3 倍的加速;而对经过特殊优化后的代码,同一个网络模型进行训练甚至会达到 3.6 ~ 3.8 倍的加速。如果读者对 GPU 的加速代码有兴趣,可以阅读 CNTK 源码(据说微软的 CNTK 拥有最好的并行化性能)。

多 GPU 的另一个好处是可以分别在每个 GPU 上运行多个模型。我们可以在不同的 GPU 上对同一个网络设置不同的参数,或者对同一个网络相同的参数输入不同的数据,从而获得更多关于该网络模型反馈信息。

总的来说,单 GPU 对所有深度学习任务来说已经足够,多 GPU 对于模型训练加速来说非常重要,毕竟没有人希望训练时间过长。

### 2. 减小 GPU 内存

在选择 GPU 时,需要根据使用到的深度神经网络模型,衡量所需内存大小,确保 GPU 有足够的内存处理该深度神经网络模型。作者在实践深度学习的初期,使用 NVIDIA GTX750 显卡训练 VGG16 网络时出现运行数分钟后报错的现象,有时甚至直接在初始化模型或者加载模型阶段报错,经过资料查明后发现是 GPU 显存不足导致。因此,GPU 的内存大小对于训练成功与否至关重要。

接下来以卷积神经网络 CNN 对于 GPU 的内存消耗为例进行讨论。因为卷积神经网络是深度学习中最为消耗内存的模型,每张图片可以有数以百万的神经元,每次卷积层可以产生成千上万的特征图,这对 GPU 内存来说是一个巨大的消耗(本节涉及很多 CNN 的知识概念,如果读者对 CNN 了解较少,建议阅读完第 4 章后重新回顾本节内容)。

在卷积神经网络 CNN 中使用反向传播 BP 算法计算卷积层的激活值和误差值的数量都是惊人的,即使网络很简单也需要大量的内存。然而精确计算在深度网络中误差值和激活值的所占用的内存是很难的。一般而言对于 ImageNet 比赛的数据集输入维度为  $224 \times 224 \times 3$  (其中  $224 \times 224$  为图像的大小,3 位通道数),在第 7 层网络的 AlexNet 网络模型中约需要 6GB 的内存。同时,网络层数越深,所需内存越多,因此减少或者降低内存的使用是很重要的技巧。

对卷积神经网络 CNN 使用更大的步长(stride),意味着我们不需要对相邻的每一个像素进行卷积,而是根据步长的大小对相隔的像素进行卷积,这样能够产生较少的输出数据,在浅层网络中经常使用该方法来减少内存的使用量。另外,还可以

使用较小的卷积核代替较大的卷积核，或者引入更多 Pooling 层。一旦我们广泛地使用了 Max Pooling、大的步长和小卷积核后，很快就会发现在这些层的处理过程中抛弃了非常多有用的信息，导致模型缺乏有效数据的支持，最后降低了模型的预测性能。即使这些技术能够有效地减少内存的消耗，但是也会对模型预测结果造成一定程度的破坏，因此需要谨慎使用。

另外我们还可以尝试修改随机梯度下降算法中 mini-batch 的大小。一个大的 batch，可以充分利用矩阵、线性代数库来加速计算，但是会增加内存消耗；batch 越小，则加速效果可能越不明显，但是内存消耗小。例如网络设置为 64 batch（每批输入 64 个样本给神经网络进行训练）会比 128 batch 减少部分内存消耗。不过带来的问题是需要更多的训练时间，特别是在训练的最后阶段，为了得到准确的梯度，稍大的 batch 明显会比小的 batch 拥有更好的表现，因此缩小 mini-batch 的大小甚至低于 8 batch，应作为最后的对策。

另一个经常被忽视的方法是改变卷积网络 CNN 所使用的数据类型，例如将 32 位换为 16 位，可以轻松帮助 GPU 节约 1/3 的内存消耗，并加快计算速度。如【代码清单 3-26】所示，可以对输入的数据 dtype 进行修改数据类型，降低计算精度。

【代码清单 3-26】修改输入数据的类型

```
from keras.layers import Input

# 修改输入数据的类型
main_input = Input(shape=(100,), dtype='int32', name='main_input')
main_input = Input(shape=(100,), dtype='int16', name='main_input')
```

## 3.4.8 训练过程的技巧

### 1. 精确率曲线和损失曲线

深度神经网络模型的训练时间比 SVM、Adaboost 等机器学习分类器的训练时间相对要长，因此训练时不应该一直等待训练结束，应持续观察精确率曲线和损失值曲线。如果发现精确率曲线不满足期望，那么可以及时停止训练，分析其原因，修改网络参数或者调整网络模型后重新训练。

如果在精确率曲线上发现训练和验证集的精确率差异越来越大（如图 3-20（a）所示），那么意味着模型可能出现了过度拟合，此时可以及时停止网络模型的训练，利用上面所学的知识向网络模型加入 L2 正则化或者添加 Dropout 层等防止过度拟合的方法。如果验证集和训练集的精确率曲线差别较少，但是两者精度都无法继续提升（如图 3-20（b）所示），那么可能意味着该神经网络模型的学习能力差，不能有

效提取输入数据的高维特征，这时可以通过对数据集进行扩展来增强数据的多样性，或者通过修改神经网络模型来增加网络中的“熵容量”，从而提高该网络的学习能力。

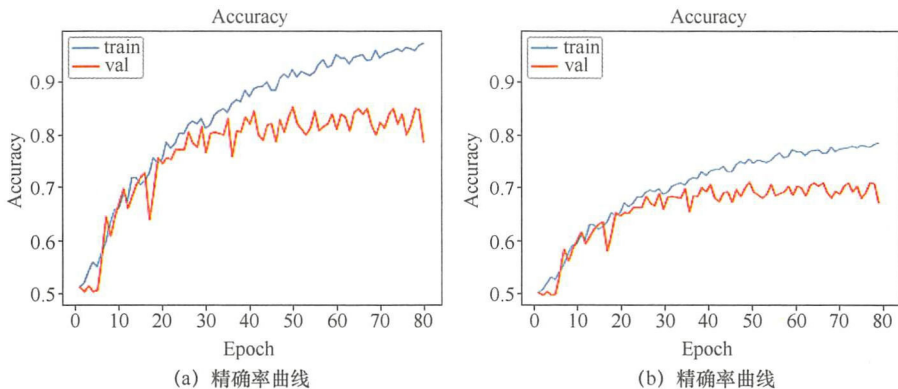


图 3-20 训练与验证的精确率曲线

对于网络训练阶段更多参数的可视化任务，可以使用英伟达公司开发的 DIGITS 工具，如图 3-21 所示。

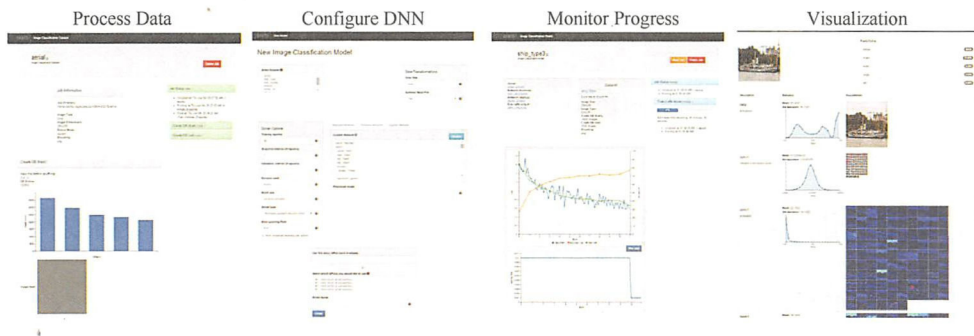


图 3-21 NVIDIA DIGITS 工具

## NVIDIA DIGITS

英伟达 (NVIDIA) 公司开发了第一个交互式深度学习 GPU 训练系统。其目的在于整合现有的深度学习开发工具，让深度神经网络设计、训练和可视化等任务变得简单化。

### 2. 网络微调 fine-tune

将预先训练好的模型权重文件的整体或者某部分用到类似的任务中，该过程称为网络模型的微调。其优点是：



- 节省训练时间；
- 当新数据集很小时，直接训练容易造成过度拟合，网络微调能够避免发生该情况。

在接下来的学习当中，部分读者或许迫切希望用自己编写的模型训练 ImageNet 数据集；或者想在使用自己的数据集，重新训练网络模型。如果有这样的想法，我们不妨考虑使用微调这一技术。

当网络模型修改预测分类时（从 1000 个分类改为 20 个分类），并不需要重新训练该神经网络，只需在修改完网络的输出层参数后微调最后几层全连接层，即可保持在之前的分类准确率的情况下得到新的模型文件。

当只增加模型的训练样本数时，只需要对浅层网络层进行一个小规模的网络微调迭代，不需要重新训练网络；当新数据集和预训练模型的数据集十分相似，并且数据集不大时，只需要对预训练模型的深层网络进行网络微调；如果数据集非常庞大，可以使用较小的学习率，对预训练模型的深层网络进行微调。

最后，如果新的数据集与原始数据的差异很大，那么网络微调的效果可能会差强人意，这时建议重新训练网络。

## 3.5 本章小结

第 2 章中简单概述了激活函数和损失函数，本章对各种类型的激活函数进行了详细的阐述，同时对损失函数进行展开，并总结了不同激活函数和损失函数的优劣。

在进一步了解深度学习之前，我们还需要掌握一些关于深度学习的规律，以便后续更方便快捷地进行实践。这些规律从数据集的准备开始，然后对数据集进行扩展，在真正使用数据时或许还需要对数据进行预处理。在对神经网络训练之前，需要仔细对网络模型进行初始化。在训练网络过程中如果出现过度拟合，可以通过正则化方法解决问题；一旦 GPU 内存不够用，可以减少内存使用量。最后需要做的就是认真观察模型损失值曲线和精确率曲线，了解模型训练阶段是否一切正常。

- 激活函数的作用：为神经网络引入非线性因素，解决线性模型不能解决的非线性问题。没有激活函数的神经网络只是一个线性回归模型，不能表达复杂的数据分布。
- 激活函数的性质：单调并且可微、限制加权输出值的范围、非线性函数。
- 常用激活函数有 Linear、Sigmoid、Tanh、ReLU、Leak ReLU、Softmax。其中 Leak ReLU 是最常用的层间激活函数，Softmax 常用作输出层激活函数。



- 损失函数：用于评价网络模型输出的预测值  $\hat{Y} = f(X)$  与真实值  $Y$  之间的差异，
- 由梯度下降算法对损失函数进行优化，更新网络中的参数。常用均方误差损失函数作为基准对其他损失函数进行评价。
- 学习率：梯度下降算法的参数。高的学习率使得误差下降快，低学习率使得误差下降平滑，一般让学习率随迭代次数分阶段衰减。
- 过度拟合：当一个模型从样本中学习到的特征不能够推广到其他新数据时，就会出现过度拟合。其出现的主要原因是训练数据中存在噪音或者训练数据过少。
- 防止过度拟合：L1、L2 正则化方法通过在损失函数中增加惩罚因子防止过度拟合，最大约束范式通过限制激活值输出防止过度拟合，Dropout 层通过减少神经元联动防止过度拟合。

## 引用/参考

- [1] Le Q V, Jaitly N, Hinton G E. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units[J]. Computer Science, 2015.
- [2] Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2013:III-1310.
- [3] Xu B, Wang N, Chen T, et al. Empirical Evaluation of Rectified Activations in Convolutional Network[J]. Computer Science, 2015.
- [4] He K, Zhang X, Ren S, et al. Identity Mappings in Deep Residual Networks[J]. 2016:630-645.
- [5] Wan L, Zeiler M, Zhang S, et al. Regularization of neural networks using dropconnect[C]// International Conference on Machine Learning. 2013:1058-1066.
- [6] Park J H, Kim Y S, Eom I K, et al. Economic load dispatch for piecewise quadratic cost function using Hopfield neural network[J]. Power Systems IEEE Transactions on, 1993, 8(3):1030-1038.
- [7] Cherkassky V, Mulier F. Statistical Learning Theory[J]. Encyclopedia of the Sciences of Learning, 1998, 41(4):3185-3185.
- [8] Bottou L. Large-Scale Machine Learning with Stochastic Gradient Descent[M]// Proceedings of COMPSTAT' 2010. Physica-Verlag HD, 2010:177-186.
- [9] Zeiler M D. ADADELTA: An Adaptive Learning Rate Method[J]. Computer Science, 2012.
- [10] Zhang S, Choromanska A, Lecun Y. Deep learning with Elastic Averaging SGD[J]. 2015:685-693.
- [11] G Chilimbi T A. Deep learning training system[J]. 2015.

# 第 4 章

## 卷积神经网络

本章主要内容：

- 卷积神经网络概述
- 卷积操作
- 三大核心思想
- 如何设计卷积神经网络

合理的损失函数加上简单的梯度下降算法，利用神经网络的模型架构就能够得到惊人的效果，这里神经网络就是深度学习的基础。到目前为止，我们已经迈进了深度学习的大门，但是诸如人工神经网络这样的全连接网络，在实际应用中不太适合处理图像任务和输入数据量过于庞大的分类任务，因为会伴随着出现过度拟合或者梯度消散等数学问题。本章要介绍的卷积神经网络（Convolutional Neural Network, CNN）能够很好地适用于图像处理、语音识别等复杂感知任务。

2012 年起，全世界最著名图像处理比赛 ImageNet 的前 10 名无一例外由基于卷积神经网络的模型所包揽。卷积神经网络模型架构经过了 5 ~ 6 年的高速发展，涌现了如 AlexNet、Google 的 GoogleNet、剑桥的 VGGNet、微软的 ResNet 等著名的网络框架，使得基于卷积神经网络进行图像分类任务的准确率足以与人类的能力相媲美。

本章将会初步介绍卷积神经网络，包括其网络结构模型和应用案例。既然是卷积网络，“卷积”操作理所当然地占据了重要地位，我们将对卷积操作的原理进行详细展开。为了更加深入地认识卷积神经网络，接着本章会讲述卷积神经网络的三大核心思想（局部感知、权值共享、下采样操作）。在进入卷积神经网络的实例之前，我们还会了解如何设计其网络架构，包括网络的层间关系和参数设计规律等。

卷积神经网络是我们踏入计算机视觉天梯的一个垫脚石，我们通过实际案例去探索其网络结构。首先我们会使用 AlexNet 网络模型对猫狗数据库进行训练和分类，并且对训练后的模型进行微调。最后我们会深入 VGGNet 网络模型的内部结构，并分解每一层卷积层进行可视化操作，深入体会其内部工作方式。

## 4.1 卷积神经网络概述

卷积神经网络的强大之处在于它的多层网络结构能自动学习输入数据的深层特征，不同层次的网络可以学习到不同层次的特征。如图 4-1 所示，浅层网络层感知区域较小，可以学习到输入数据的局部域特征（如图像物体的颜色、几何形状等）；深层网络层具有较大的感知域，能够学习到输入数据中更加抽象一些特征（如图像物体的属性、轮廓特点、位置信息等高维性质）。深层次的抽象特征对图像中物体的大小、位置和方向等敏感度较低，从而大大提高了物体的识别率，因此卷积神经网络常用于图像处理领域。



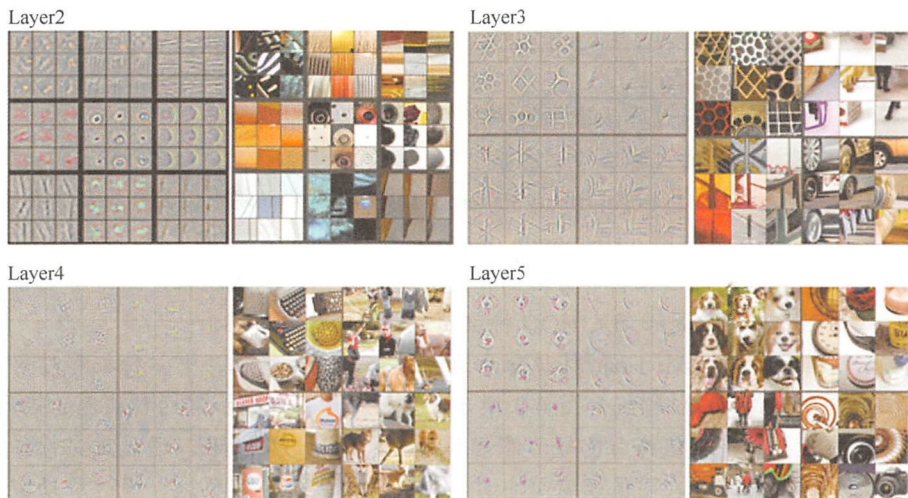


图 4-1 可视化卷积核。Layer 数值越大，表示网络的层数越深

### 4.1.1 卷积神经网络的应用

卷积神经网络为图像而生，但应用不限于图像。在图像处理任务上，卷积神经网络可以用来识别位移、缩放及物体形态扭曲的二维图形。一方面，由于其网络模型中的特征是通过训练数据集进行图像特征学习，从而避免了显式地特征抽取；另一方面，由于图像上同一特征映射面上的神经元权值相同，所以卷积神经网络模型可以并行训练，极大地提高神经网络的训练时长。此外，与神经元彼此相连的神经网络（如传统的人工神经网络）相比，卷积神经网络模型的组织方式特殊，其结构模型更易于理解和分析。

卷积神经网络的应用场景和案例并不一定能够真正应用在实际工程领域，但也是足够精彩，因为它不仅代表业界最先进的视觉技术（state-of-the-art），甚至还可能超出我们的想象范围，下面就了解卷积神经网络的具体应用。

#### 1. 图像分类与识别

在卷积神经网络还没有普及之前，通常由人工抽取图像中的特定信息（如轮廓检测，边缘检测，LBP、HOG、HAAR 等特征检测方法）来实现图像分类任务，然后对这些特征编写特定的算法来对分类模式进行匹配。如此显式地抽取图像特征的方法，不仅在特征工程问题上耗费了工程师们大量的时间，而且仍然会存在着许多严峻的问题等待工程师们去解决。如图像受光照影响、物体旋转影响、物体平移等空间信息的改变，其图像中物体的特征也会随之改变等，从而导致之前的模式识别



方法失效。

传统图像检测特征方法：

- 边缘检测：通过算法检测图像中明暗变化剧烈的像素点。
- 轮廓检测：通过算法检测图像中目标物体的边界信息。
- 局部二值模式（Local Binary Pattern, LBP）：用来描述图像局部特征的算子。
- 方向梯度直方图（Histogram of Oriented Gradient, HOG）特征检测：HOG 特征通过计算和统计图像局部区域的梯度方向直方图来构成特征，通过滑动窗口的方式在图像中检测是否满足 HOG 特征的窗口作为检测目标。
- Haar 特征检测：Haar 特征分为边缘特征、线性特征、中心特征和对角线特征，组合而成的特征模板如图 4-2 所示。该特征模板有白和黑两种颜色的矩形，并定义该模板的特征值为白色矩形像素和减去黑色矩形像素和。通过改变特征模板的大小和位置，可在图像子窗口中穷举出大量的特征。同样地，通过滑动窗口的方式在图像中检测是否满足 Haar 特征的窗口作为检测目标。



图 4-2 Haar-Like 特征

在 2014 年的 ImageNet 图像分类比赛上，AlexNet 网络模型大幅度地超越了其他选手，夺得了当年图像分类大赛的冠军，因为 Alex 使用并改进了卷积神经网络模型。从那以后，卷积神经网络在图像分类上一枝独秀，其中手写字体（Hand Written）的识别率已经超越人类的识别率，达到了 99.9%。国外众多快递公司已经开始应用卷积神经网络模型识别快递单上的手写字体，尽最大可能地节约企业成本、提高自身的系统运作效率。

完成图像分类之后，更加富有挑战性的工作是对整体图像进行目标识别。因为一般图像中不只有一个类别，例如一张图片中可能包括多个类别：一只狗、一栋房子、一棵树等。

近年来自动驾驶和辅助驾驶抢占了各大媒体的头条，正是因为卷积神经网络的帮助，它能够对车载终端采集到的图像进行强大的感知和处理（如图 4-3 所示）。

## 2. 自然语言处理 NLP

卷积神经网络不再是图像处理任务专用的神经网络模型。近两年来，学者们将卷积神经网络应用于自然语言处理（Natural Language Processing, NLP）领域的研究，已经有了十分出色的表现，新成果和顶级论文层出不穷。

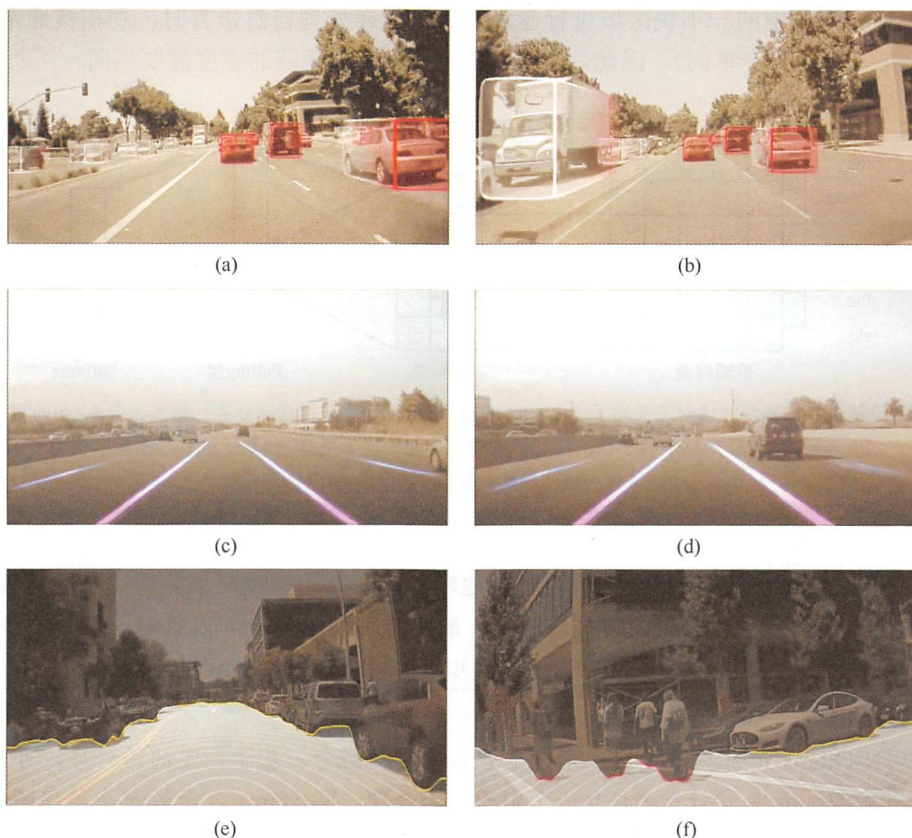


图 4-3 卷积神经网络应用于自动驾驶领域，应用场景为高速公路和城市道路。其中，(a) (b) 为使用卷积神经网络进行前车三维大小识别，(c) (d) 为使用卷积神经网络进行车道线识别与跟踪，(e) (f) 为使用卷积神经网络检测视频中可行驶道路范围。使用卷积神经网络在自动驾驶领域当中的算法可以视为使用摄像头对路面的情况进行感知，从而获取行驶当中所需的信息

使用卷积神经网络进行自然语言处理主要是针对语义分析和话题分类两大任务。根据 (KIM, Yoon et al., 2014) 中的网络模型 (如图 4-4 所示)，共分为 4 层，输入层是一个表示句子的矩阵，每一行代表一个单词向量，输入层后接一个卷积层和 Pooling 层，最后是 Softmax 分类器作为输出层。令人惊讶的是，如此简单的卷积神经网络模型在各个自然语言处理的公共数据集上的表现都非常出色，个别语言数据库甚至刷新了目前最好的自然语言处理算法结果。

自然语言处理任务在卷积神经网络模型中的输入不再是像素点，大多数情况下是以矩阵表示的句子。矩阵的每一行对应一个元素，如果一个元素代表一个单词，那么每一行代表一个单词的向量。卷积神经网络模型应用在计算机视觉中，卷积核

每次只对图像中的一小块区域进行卷积操作，但在处理自然语言时，卷积核通常覆盖上下几行（几个单词）。因此，卷积核的宽度和输入矩阵的宽度需要相同。

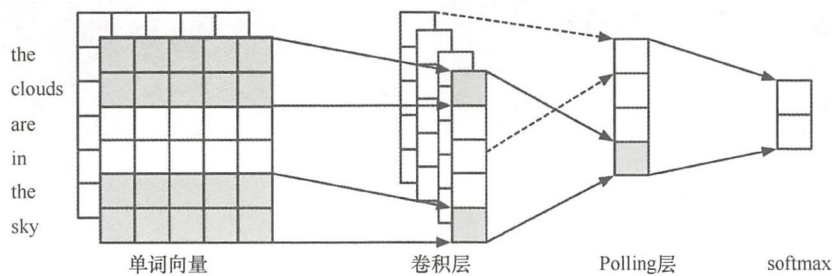


图 4-4 使用卷积神经网络对英文句子“the clouds are in the sky”进行话题分类。输入一个 6×5 的矩阵，每一个单词当作矩阵中的一行，经过一次卷积层和一次下采样 Pooling 层后，输出层使用 Softmax 作为激活函数，最终得到两个输出预测分类

相比于 N-Grams，卷积神经网络对特征表示方式的效率更胜一筹。由于词典庞大，超过 3-Grams 后计算开销就会急剧地增大，而卷积神经网络能够轻松地处理超过 3-Grams 的计算量。更值得关注的是，最新的研究（Poznanski et al., 2016）表明在卷积神经网络第一个卷积层中滤波器感知到的特征与 N-Grams 非常相似。

**N-Grams**

N-Grams 是自然语言处理中一个非常重要的概念，基于一定的语料库，可以利用 N-Grams 来预计或者评估一个句子是否合理。N-Grams 表示第  $N$  个单词的出现只与前面  $N-1$  个单词相关，与其他不在该范围的任何单词都没有关联，整句的概率就是各个词出现概率的乘积。

### 3. 图像着色

图像着色问题是指将颜色添加到灰度图像中，即灰度图像恢复色彩的过程。传统的做法是人工去对每一帧图像中的每一个像素和每一个物体进行着色，这是一项艰巨的任务。使用人力手工完成该任务会带来两个大问题：

- （1）耗费大量的人力资源和宝贵的时间；
- （2）对于同一个事物，不同人的着色标准是有差异的。

（Zhang et al., 2016）使用改进的卷积神经网络模型提取输入图像的特征及其上下文信息来对图像进行着色，最终效果如图 4-5 所示。以后，年代久远的纪录片可以真实还原当年的色彩，父母的老照片也终于可以还原出当年色彩艳丽的场景。

卷积神经网络的应用当然不止于此，我们还可以利用它进行人体姿态估计、动作跟踪、视频帧预测、视频内容分类、视频标注等（Kenshimov et al., 2017）。





图 4-5 使用卷积神经网络模型对灰度图进行着色的结果，从左到右分别为：原始输入的  
黑白图像、正在迭代的着色图片和最终产生的彩色图片

卷积神经网络以其局部权值共享的特殊结构在语音识别、自然语言处理、图像处理方面有着独特的优越性。因为该网络模型的布局相对于人工神经网络更接近实际的生物神经网络，权值共享降低了网络的复杂性。特别是在图像处理领域，图像可以直接作为卷积神经网络模型的输入，避免了特征提取的特征工程和特征分类的模式识别过程，因此卷积神经网络应用比传统的神经网络更为广泛，也为深度学习的崛起发挥着巨大的作用！

### 4.1.2 卷积神经网络的结构

卷积神经网络主要由卷积层、下采样层、全连接层 3 种网络层构成（这里没有包含输入层和输出层）。上述 3 种网络层经过排列组合，就可以构建一个完整的卷积神经网络（如图 4-6 所示）。

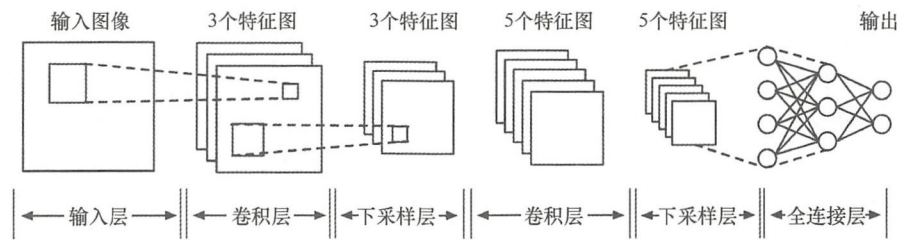


图 4-6 卷积神经网络的简单示例模型，该卷积神经网络的网路层从左到右分别为：输入层、  
第一个卷积层、第一个下采样层、第二个卷积层、第二个下采样层、全连接层、输出层

为了更好地理解卷积神经网络，下面以图 4-6 所示的卷积神经网络为例简单介绍卷积神经网络的组成架构。这里我们暂且不管卷积层和下采样层的具体操作，同时假设图像经过卷积层后的输出特征图大小与输入时的大小一致，把重心放在卷积神



经网络的网络结构的组织上。

假设图 4-6 中的输入图像为灰度图  $A$ ，对应图像矩阵表示为  $(1, w, h)$ ，1 为单通道， $w$  和  $h$  对应输入图像的长和高。首先把灰度图  $A$  作为输入层，然后接一个卷积层。第一个卷积层对输入的图像进行卷积操作后，得到 3 个特征图（Feature Map），每个特征图对应一个卷积核，此时网络模型中的数据存储结构变成  $(3, w, h)$ 。因此 3 个特征图组成的矩阵也被称为“特征矩阵”，这里的 3 代表特征矩阵的深度。值得注意的是，卷积层操作产生多少个特征图是自由设定的，也被称为超参数（Hyper-Parameters）。

第一个卷积层后接的是下采样层（下采样层一般称为 Pooling 层，后续章节我们会统一称为 Pooling 层），Pooling 层对输入的 3 个特征图（特征矩阵）进行下采样操作，得到 3 个更小的特征矩阵  $(3, w/2, h/2)$ 。一般来说，对图像进行下采样操作得到的特征图作为输入 Pooling 层特征图的一半。

接下来到第二个卷积层，卷积操作之后产生 5 个特征图  $(5, w/2, h/2)$ 。然后再接一个 Pooling 层，对 5 个特征图进行下采样操作，得到 5 个更小的特征图  $(5, w/4, h/4)$ 。

在第二个 Pooling 层后面接的是两个全连接层。第一个全连接层的每个神经元与上一层的 5 个特征图中每个神经元（每个像素）进行全连接。下一个全连接层同样与上一层的每个神经元进行全连接。

在整个网络的最后一层是输出层（Softmax 层），对全连接后的特征向量进行计算，得到分类评分值。与普通神经网络类似，卷积神经网络模型中的卷积核都是通过对输入数据集经过梯度下降算法训练得到的。

通过上述例子介绍卷积神经网络，我们了解到卷积神经网络最简单的架构方式，及其数据的传递方式。输入一张图像到卷积神经网络中，经过若干卷积层和 Pooling 层，对图像进行计算操作得到特征图后传递给下一层，最后经过全连接网络输出该图像的分类评分结果。

下面我们逐层展开介绍卷积神经网络的网络结构，图 4-7 以图 4-6 的网络模型为例：输入层→卷积层→Pooling 层→卷积层→Pooling 层→全连接层→全连接层→Softmax 层。对图 4-6 进一步展开，从图 4-7 中我们可以清晰地看到图像输入到卷积神经网络模型后其数据的流动方向。

### 1. 输入层（Input Layer）

输入图像，假设输入图像为  $[1, 32, 32]$  的灰度图，当然也可以输入彩色图。

### 2. 卷积层（Convolution Layer）

对输入卷积层的图像或者特征图进行卷积操作，输出卷积后的特征图。图 4-7 中定义了第一个卷积层有 3 个卷积核，因此卷积操作后得到的特征矩阵为  $[3, 32, 32]$ ；定义第二个卷积层有 5 个卷积核，同理卷积操作后得到的特征矩阵为  $[5, 32, 32]$ 。

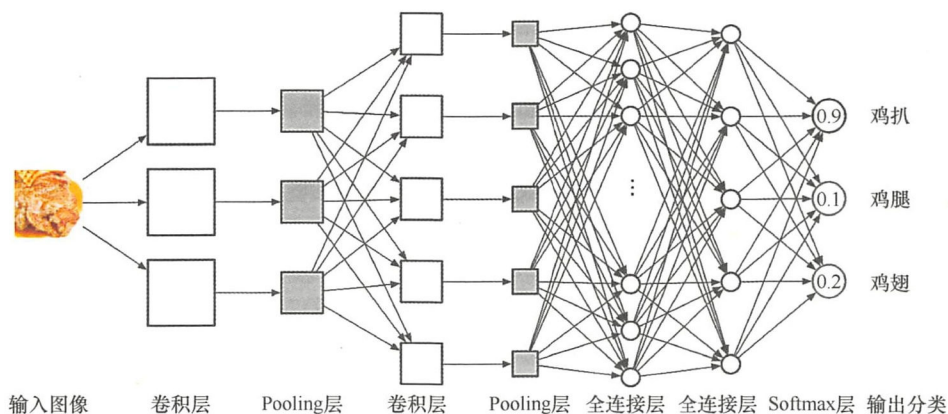


图 4-7 卷积神经网络的结构模型。输入一张图片，经过卷积神经网络模型后输出分类的结果，其中方框代表图像矩阵，圆形代表神经元，箭头代表数据的流动方向

### 3. 下采样层 (Polling Layer)

Pooling 层对传入的图像在空间维度上进行下采样操作，使得输入的特征图长和宽均变为原来的一半，本例中第一次卷积层后得到  $[3, 32, 32]$  大小的特征矩阵作为 Pooling 层的输入，输出  $[3, 16, 16]$  大小的特征矩阵，再经过第二个 Pooling 层后输出则变成  $[5, 8, 8]$  大小的特征矩阵。

### 4. 全连接层 (Fully Connected Layer)

全连接层与普通神经网络一样，每个神经元都与输入的所有神经元相互连接，然后经过激活函数进行计算。图 4-7 中最后 Pooling 层得到  $[5, 8, 8]$  大小的特征向量，即一共有  $5 \times 8 \times 8 = 320$  个神经元，假设全连接层的神经元为 100，那么全连接产生  $320 \times 100 = 32000$  条连接线，后面的全连接层与 ANN 类似。

### 5. 输出层 (Output Layer)

输出层有时也被称为分类层，因为在最后输出时，将会计算每一类别的分类评分值。假设输出的图像分类选项为（鸡扒、鸡腿、鸡翅），那么输出层的输出为  $[1, 3]$  大小的矩阵，最终的输出结果为  $[0.9, 0.1, 0.2]$ ，输出层选择概率最高的“鸡扒”作为本次卷积神经网络的图像分类任务的预测结果（对应类别的预测概率）。

从上述的网络结构可以看出，卷积神经网络模型逐层对图像的每一个像素值进行计算，到最后经过卷积神经网络模型后，输出分类评分值。

## 4.1.3 卷积神经网络与人工神经网络的联系

人工神经网络使用的是全排列的方式，即神经元按照一维进行排列；而卷积神

神经网络每层神经元都是按照三维排列，每一层有其长、宽、高。其中长、宽代表输入图像矩阵的长度和高度，宽代表该层网络的深度。虽然人工神经网络和卷积神经网络有着各自的排列方式，但是其内部实现原理都是由神经元模型组成的神经网络。

如图 4-8 下半部分所示为卷积神经网络结构中的卷积层，上半部分为其对应的神经网络示例图。卷积层有  $n$  个卷积核对应输出有  $n$  个特征图：假设卷积层输入为一张  $4 \times 4$  大小的图像矩阵（对应输入神经元有  $16 (4 \times 4)$  个），设定卷积核为 2，经

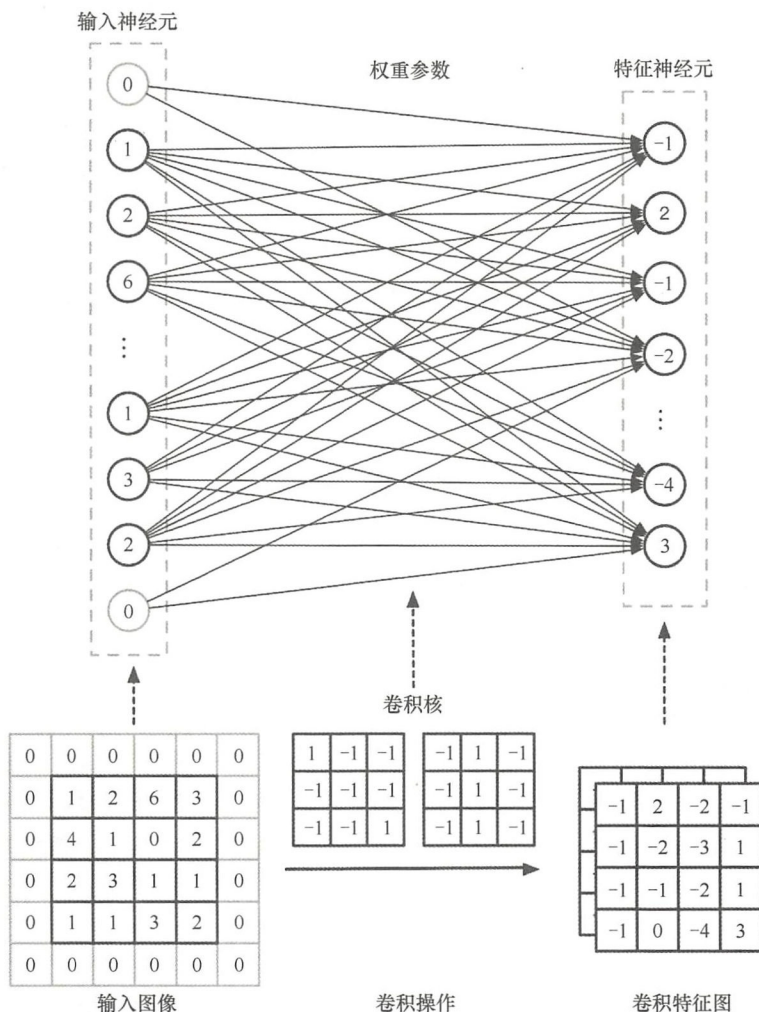


图 4-8 卷积神经网络的网络结构中，卷积层后接 Pooling 层。输入为  $4 \times 4$  大小的图像矩阵，卷积层参数为 ( $\text{padding}=\text{same}$ ,  $\text{kernel}=2$ )，卷积层后输出特征矩阵大小为  $[2, 4, 4]$



过卷积操作后产生2个  $4 \times 4$  大小的特征图（对应输出特征神经元有  $32 (2 \times 4 \times 4)$  个）。其中，卷积核大小为  $3 \times 3$ （一个卷积核由 9 个神经元组合而成），即对应的权重参数连接线有  $18 (2 \times 3 \times 3)$  条，输入神经元与特征神经元连接线均带有权值参数  $w_i$  和偏置  $b_i$ ，而该权值参数则由卷积核组成。

卷积神经网络的网络模型的卷积层和全连接层中的权重参数经过梯度下降算法（SGD）进行训练得到，最终使得卷积神经网络计算出的分类概率能和训练集中的图像标签吻合。

## 4.2 卷积操作

卷积操作实际上是图像处理技术中的滤波操作，因此卷积核又被称为滤波器，在这里我们统称为“卷积核（Convolution Kernel）”。不同于传统的滤波操作，滤波器是事先定义好的，而卷积神经网络的卷积核内容则是通过梯度下降算法（SGD）训练得到的。

### Padding 操作

卷积操作时会遇到在图像边界卷积造成图像信息丢失的问题，而 Padding 操作则是为了解决边界卷积问题而提出的，这里简单介绍 Padding 操作的工作原理。

Padding 操作可以分为 Same Padding、Valid Padding 和自定义 Padding 3 种方法。Same Padding 是根据卷积核大小，对输入图像矩阵进行边界补充（一般填充零值），使得卷积后得到的特征矩阵与输入矩阵大小一致；Valid Padding 实际上不需要进行 Padding 操作；而自定义 Padding 生成的特征图大小根据下式计算而来：

$$\begin{aligned} output_h &= (input_h + 2 \times padding_h - kernel_h) / stride + 1 \\ output_w &= (input_w + 2 \times padding_w - kernel_w) / stride + 1 \end{aligned} \quad (4-1)$$

式（4-1）中， $output$  为输出矩阵大小， $input$  为输入矩阵大小， $padding$  为边界填充数量， $kernel$  为卷积核大小， $stride$  为步长大小；另外下标  $w$  为操作矩阵的宽， $h$  为操作矩阵的长。

图 4-9 所示为 Padding 操作的两种主要方式，图中输入为  $5 \times 5$  大小的图像矩阵，卷积核大小均为  $3 \times 3$ 。图 4-9（a）Same Padding 方法设置  $padding$  参数为 1，使得卷积的输出与输入矩阵大小一致。图 4-9（b）Valid Padding 方法设置  $padding$  参数为 0，使得输出特征矩阵比输入矩阵要小。

本章中如果没有特殊声明，一般默认卷积操作中使用 Same Padding 方法，也就是对输入图像矩阵的边缘填充零像素值，使得输入的图像经过卷积后得到的特征矩阵大小与输入的原图大小一致。默认使用 Same Padding 的好处是可以避免边界信息



被忽略，把边界信息也纳入神经网络的计算范围内，否则随着神经网络层的深入，图像边缘信息的损失会逐渐增大。

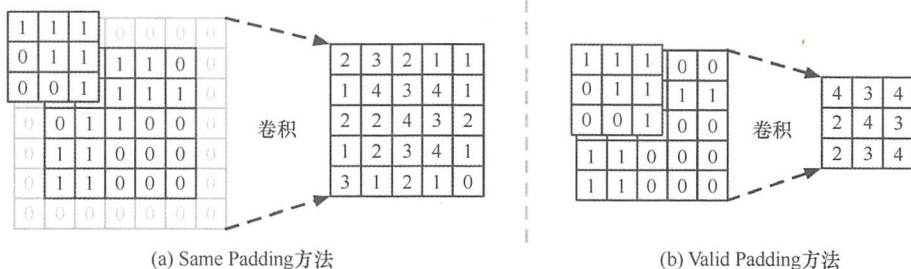


图 4-9 Padding 操作。(a) 为 Same Padding 操作，对原始输入矩阵的上下左右 4 个边均添加了一行为零的像素边缘，在使用  $3 \times 3$  卷积核的前提下，卷积操作后得到的特征图与输入矩阵大小一致；(b) 为 Valid Padding 操作，也就是直接对原始数据矩阵进行卷积，卷积后得到的特征图尺寸比原始输入矩阵要小

## 4.2.1 滑动窗口卷积操作

在卷积操作的过程中，原始输入为一张图像，通过一个卷积核在输入图像上进行滑动，对滑动窗口与卷积核进行数值运算后，卷积后输出的为特征图。卷积神经网络中的卷积核由权重参数  $w$  组成，当网络中的权重参数改变时，意味着使用另外一个卷积核，生成另一个特征图。

假设一个卷积核窗口在输入图像上滑动，滑动窗口每次移动的步长为  $stride$ ，那么每次滑动窗口后，把求得的值按照空间顺序组成一个特征图。该特征图的边长分别为：

$$\begin{aligned} feature\ map_w &= (img_w - kernel_w) / stride + 1 \\ feature\ map_h &= (img_h - kernel_h) / stride + 1 \end{aligned} \quad (4-2)$$

在式 (4-2) 中， $feature\ map$  为特征图矩阵的大小， $img$  为输入图像大小， $kernel$  为卷积核大小， $stride$  为步长大小， $w$  为矩阵的宽， $h$  为矩阵的长。

图 4-10 为单次卷积操作的过程，图中使用  $3 \times 3$  的卷积核对  $5 \times 5$  的图像进行卷积操作。图像虚线下方部分， $3 \times 3$  大小的卷积核首先对应到输入图像左上角位置的  $3 \times 3$  大小窗口，然后使用卷积核里的数值与滑动窗口里的像素值一一相乘，接着对相乘后的矩阵进行总体求和，其和作为原图在该位置的卷积特征值。

使用同一个卷积核和图像矩阵作为输入，剩余卷积过程如图 4-11 所示。假设本次卷积的步长  $stride=1$ ，那么第二次卷积则是把滑动窗口向右移动一格，重复卷积操作，得到卷积特征 3。不断重复该过程，直至滑动窗口移动到输入矩阵的右下角为

止，得到由卷积特征组成的特征图。

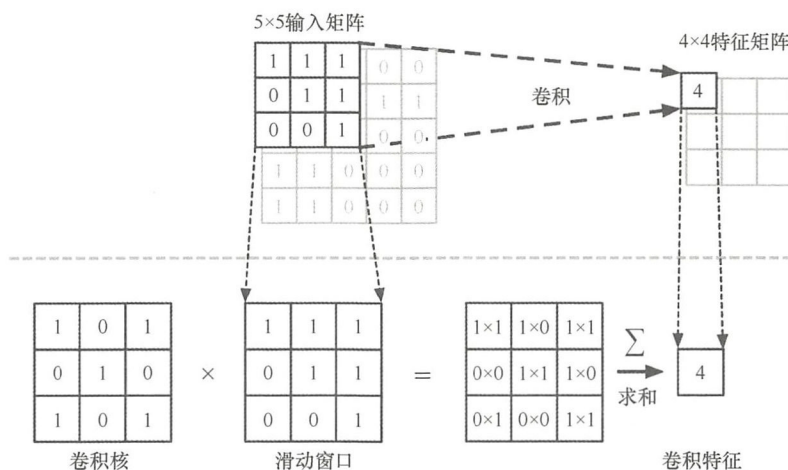


图 4-10 一个 3×3 大小的卷积核对一张 5×5 大小的图像矩阵进行卷积操作。其中，中央虚线为分割线，虚线上部分为单次卷积操作，虚线下部分为对该单词卷积操作进行展开，通过矩阵相乘、求和操作后得到单个滑动窗口的卷积特征 4

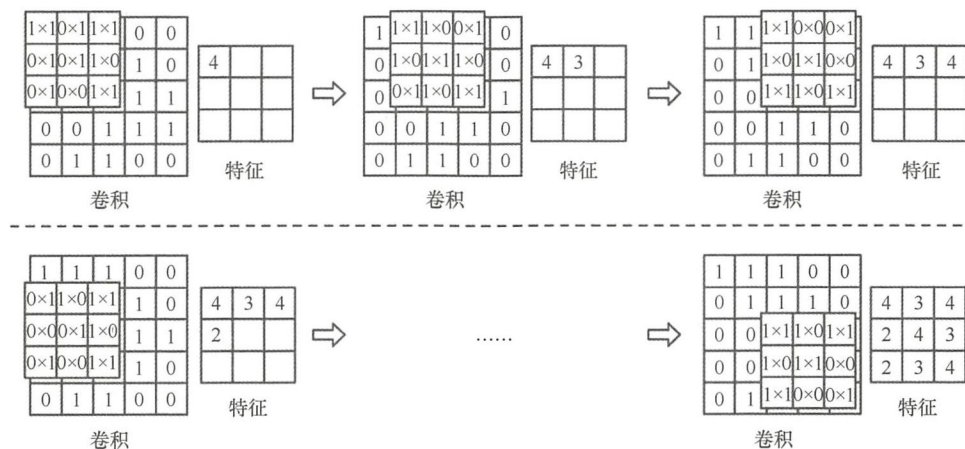


图 4-11 卷积操作。使用 3×3 的卷积核，对 5×5 大小的图像进行卷积的全过程。顺序为从左到右、从上到下，经过图 4-10 所示的卷积操作，最后填满卷积后的特征矩阵

## 4.2.2 网络卷积层操作

上面我们介绍了单个输入矩阵的卷积操作。那么卷积神经网络的层与层之间是

如何进行卷积操作的呢？该问题可以转化为：在第  $l$  层网络有  $C$  个特征图作为输入，该卷积层有  $k$  个卷积核，如何进行卷积操作产生  $k$  个特征图作为输出呢？

假设在第  $l$  层卷积层输入为  $C$  个特征图，即该层输入  $C$  个矩阵， $C$  个矩阵的大小均为  $W \times H$ ，那么我们可以得到一个  $C \times (W \times H)$  的特征张量， $C$  又称为输入矩阵的深度。该层设定有  $C_{out}$  个  $K \times K$  大小卷积核，在使用 Same Padding 的情况下将会产生  $C_{out}$  个大小为  $W \times H$  的特征图作为输出，即可以得到  $C_{out} \times (W \times H)$  的特征张量作为输出。

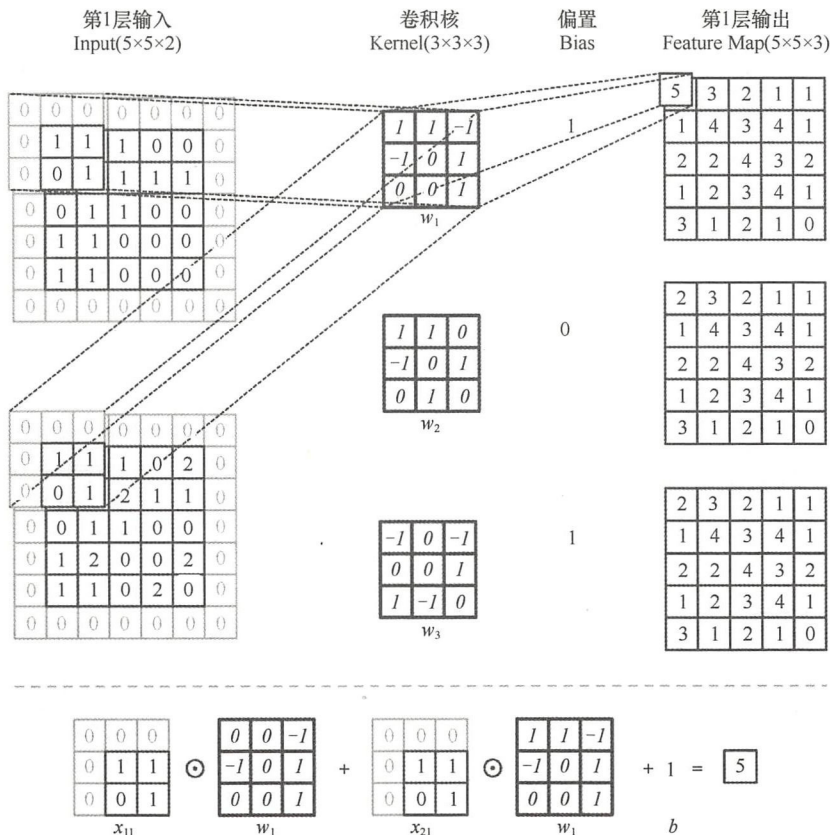


图 4-12 卷积神经网络的层间卷积操作。 $l$  层输入为两张大小为  $5 \times 5$  的特征，第  $l$  个卷积层有 3 个卷积核，因此输出有 3 张特征图，作为  $l+1$  层的输入

接着我们分析图 4-12 的具体计算过程，即第一个特征图的第一个输出结果 5 是如何计算得到的。其实，多层卷积操作类似于神经元的基本求和公式  $z = \sum w x + b$ ，卷积神经网络中  $w$  对应单个卷积核， $x$  为对应输入矩阵的不同数据窗口， $b$  为该卷积

核的偏置。这相当于卷积核与一个个数据窗口相乘求和后（矩阵内积计算），加上偏置  $b$  得到输出结果。该过程如下：

$$\begin{aligned} z &= \mathbf{x}_{11} \odot \mathbf{w}_1 + \mathbf{x}_{21} \odot \mathbf{w}_1 + b \\ &= (\mathbf{x}_{11} + \mathbf{x}_{21}) \odot \mathbf{w}_1 + b \\ &= 4 + 1 \\ &= 5 \end{aligned}$$

在上式中， $z$  为第一个特征图的第一个输出结果， $\mathbf{w}_1$  为第一个卷积核， $\mathbf{x}_{11}$  为第一个输入矩阵的第一个  $3 \times 3$  的窗口， $\mathbf{w}_{21}$  为第二个输入矩阵的第一个  $3 \times 3$  窗口。

接下来卷积核  $\mathbf{w}_1$  固定不变，数据窗口统一向右移动 1 步（ $\text{stride}=1$ ）。重复上述步骤对矩阵进行内积计算，直到数据窗口滑动到输入矩阵的右下角，卷积核  $\mathbf{w}_1$  得到其对应的输出特征图。依此类推，对应第二个卷积核  $\mathbf{w}_2$ ，同样重复上述步骤得到对应第二个输出特征图。

最后使用激活函数对得到的 3 张输出特征图经过激活函数计算  $a=f(z)$ ，作为第  $l$  层卷积层的最终输出特征图。

### 4.2.3 矩阵快速卷积

卷积操作是在图像中通过滑动窗口，逐像素进行矩阵计算，会耗费大量的计算资源去寻址和修改内存数据，因此最终的卷积操作并不是如上一节中的滑动窗口方法进行卷积操作，而是采用转换为矩阵的方式进行快速计算。矩阵操作能在计算机中快速运算并且方便移植到 GPU 中，在实际生产环境中可以通过两步来完成卷积操作：

（1）使用 Image to column（Im2col）算法把输入图像和卷积核转换成为规定的矩阵排列方式；

（2）使用 GEMM 算法对转换后的两个矩阵进行相乘，得到卷积结果。

如图 4-13 上半部分所示为对输入图像矩阵进行 Im2col 算法操作。假设输入的图像大小为  $C \times H \times W$ （其中  $H$  为图像的长， $W$  为图像的宽， $C$  为图像的深度）。卷积核的大小为  $K \times K$ ，那么对应输入图像中一个卷积窗口可以表示为  $C \times (K \times K)$  的向量，即对输入图像中的某位置的数据按照卷积窗口进行重新排列，得到  $C \times (K \times K)$  的特征向量。

按照上述原理，以步长为 1（ $\text{stride}=1$ ）从输入图像的左上角开始对原图进行特征转换，最终得到特征图大小为  $(H \times W) \times (C \times K \times K)$ 。

图 4-13 下半部分为卷积核的 Im2col 操作。假设有  $C_{out}$  个卷积核，每个卷积核大小为  $C \times (K \times K)$ ，把卷积核进行矩阵变换，得到单个卷积核的尺寸为  $C \times K \times K$ 。依此类推，最终得到  $C_{out} \times (C \times K \times K)$  大小的过滤矩阵（Filter Matrix）。



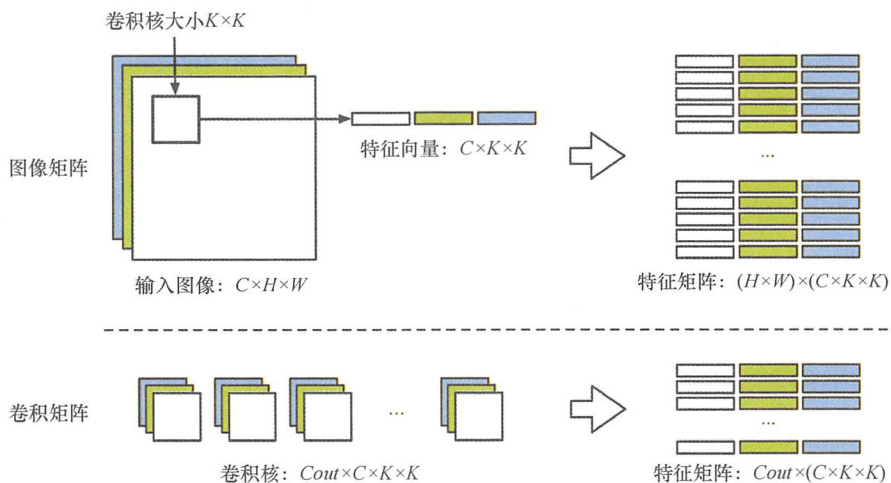


图 4-13 Im2col 算法操作。图中虚线为分割线，虚线上部分为图像矩阵的 Im2col 操作，左为  $C \times H \times W$  大小的图像，右为转换后特征矩阵，大小为  $(H \times W) \times (C \times K \times K)$ ，两矩阵保持总参数大小不变；

下部分为卷积矩阵的 Im2col 操作，作为  $C_{out}$  个  $C \times K \times K$  大小的卷积核，右为大小为  $C_{out} \times C \times K \times K$  大小的过滤矩阵，同样两矩阵保持总参数大小不变

一般矩阵乘法（General Matrix Matrix Multiply, GEMM）将由卷积核产生的过滤矩阵乘以原图产生的特征图矩阵（Feature Matrix）的转置，得到大小为  $C_{out} \times (H \times W)$  的输出特征图矩阵：

$$\begin{aligned}
 \text{feature map} &= \text{Filter Matrix} \cdot \text{Feature Matrix}^T \\
 &= [C_{out} \times (C \times H \times W)] * [(C \times H \times W) \times (H \times W)] \quad (4-3) \\
 &= C_{out} \times H \times W
 \end{aligned}$$

假设卷积核为  $2 \times 2$  的矩阵，输入原图像 Image 为  $3 \times 3$  的单通道矩阵，边界扩展为 0，滑动步长为 1（padding=0, stride=1, c=1, w=3, h=3, k=2）。因为 padding=0，因此过滤矩阵的长宽 w、h 均减少了 1，经过 Im2col 变换后特征图矩阵的大小为  $(2 \times 2) \times (1 \times 2 \times 2)$ ：

$$\text{Image} = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 1 & 2 \\ 3 & 1 & 1 \end{bmatrix} \Rightarrow \text{Feature Matrix} = \begin{bmatrix} 3 & 2 & 0 & 1 \\ 2 & 1 & 1 & 2 \\ 0 & 1 & 1 & 3 \\ 1 & 2 & 1 & 1 \end{bmatrix}$$

假设有 2 个大小均为  $2 \times 2$  的卷积核 A、B (cout=2, c=1, k=2)，因此过滤矩阵大小为  $2 \times (1 \times 2 \times 2)$ ：

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}, B = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \Rightarrow \text{Filter Matrix} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \end{bmatrix}$$

输出的特征矩阵  $C$  为 *Filter Matrix* 乘以 *Feature Matrix*<sup>T</sup> :

$$\begin{aligned}
 C &= \text{Filter Matrix} \cdot \text{Feature Matrix}^T \\
 &= \begin{bmatrix} 0 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \end{bmatrix} \times \begin{bmatrix} 3 & 2 & 0 & 1 \\ 2 & 1 & 1 & 2 \\ 0 & 1 & 1 & 3 \\ 1 & 2 & 1 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} 0 & 1 & 1 & 2 \\ 2 & 1 & 1 & 3 \end{bmatrix} \\
 &\Rightarrow \left[ \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \right]
 \end{aligned}$$

其中, 特征矩阵  $C$  中的  $\begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$  和  $\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$  分别为输出的两个特征图。

【代码清单 4-1】为 Im2col 的 Python 实现, 其中分为 get\_im2col\_indices 函数和 im2col\_indices 函数。

#### 【代码清单 4-1】Im2col 具体实现代码

```
def get_im2col_indices(x_shape, field_height, field_width, padding=1, stride=1):
    # 最后输出图的矩阵大小
    N, C, H, W = x_shape
    assert (H + 2 * padding - field_height) % stride == 0
    assert (W + 2 * padding - field_width) % stride == 0

    # out_width 卷积核在输入图像上滑动横向可截取的最大特征数
    # out_height 卷积核在输入图像上滑动竖向可截取的最大特征数
    out_height = (H + 2 * padding - field_height) / stride + 1
    out_width = (W + 2 * padding - field_width) / stride + 1

    i0 = np.repeat(np.arange(field_height), field_width)
    i0 = np.tile(i0, C)
    i1 = stride * np.repeat(np.arange(out_height), out_width)
    j0 = np.tile(np.arange(field_width), field_height * C)
    j1 = stride * np.tile(np.arange(out_width), out_height)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)

    k = np.repeat(np.arange(C), field_height * field_width).reshape(-1, 1)
    return (k, i, j)
```

```
def im2col_indices(x, field_height, field_width, padding=1, stride=1):
    """ im2col 的python实现 """
    # Zero-pad the input
    p = padding
    C = x.shape[1]
    x_padded = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant' )

    k, i, j = get_im2col_indices(x.shape, field_height, field_width, padding, stride)

    cols = x_padded[:, k, i, j]
    cols = cols.transpose(1, 2, 0).reshape(field_height * field_width * C, -1)
    return cols
```

## 4.3 卷积神经网络三大核心思想

卷积神经网络并不复杂，相比人工神经网络，卷积神经网络在性能和应用场景上带来了极大的提升。

在功能上，卷积神经网络引入了三大核心思想：

- 局部感知；
- 权值共享；
- 下采样技术。

卷积神经网络的三大核心思想使得卷积神经网络能够提取图像物体的高维特征，感知图像中更丰富的信息。同时，经过权值共享和下采样操作，进一步减少网络的参数，让卷积神经网络模型能够在规定时间内和有限的内存硬件下完成计算。

在性能上，原来图像中基于滑动窗口的卷积操作，实际上可以使用矩阵来快速卷积代替，这极大地减少了卷积运算的时间。此外，GPU 强大的并行能力可以让庞大的数据量分布式并行计算，使得庞大的参数运算不再成为硬件性能的瓶颈。

### 4.3.1 传统神经网络的缺点

卷积神经网络并没有采用如人工神经网络的全连接方式，而是以图像矩阵的方式进行排列，原因如下。

#### 1. 庞大的参数

图像在卷积神经网络中每个像素值代表一个神经元节点，一张  $400 \times 400$  的图像作为输入，那么在输入层会产生  $400 \times 400 = 1.6 \times 10^5$  个输入节点。假设卷积神经网络

有 3 个隐层，每一层有 100 个节点，全连接后大约产生  $400 \times 400 \times 100^3 = 1.6 \times 10^{11}$  个权重参数，庞大的参数对于昂贵的计算资源来说是不可接受的。

## 2. 丢失像素间信息

图像实际上是由数据间相互有关联的矩阵组成的，因此图像中的相邻像素间有着紧密的联系，一个像素值代表一个神经元节点，那么就会大量地丢失像素间相邻的关系信息。

## 3. 制约网络深度发展

通常来说，神经网络的层数越深，提取的数据维度越高，当全连接网络超过 3 层后，很容易引发数据过度拟合、梯度消散等一系列数学问题，所以传统神经网络的隐层数量设置一般不建议超过 3 层。

卷积神经网络的神奇之处就在于能够用自身独特的方式避免传统神经网络所引起的问题，其优点如下。

- 局部感知：每一个神经元节点不再与下一层的所有神经元节点相连接（全连接的方式），只与下一层的部分神经元进行连接。
- 权值共享：一组连接可以共享同一个权重参数，或者多组连接共享同一个卷积核，不再是每条连接都有自己的权重。
- 下采样 Pooling：通过 Pooling 技术对输入的数据进行下采样压缩操作，减少输出节点。

卷积神经网络通过局部感知和权值共享，保留了像素间关联信息，并且大大减少了所需参数的数量。通过 Pooling 技术，进一步缩减网络参数数量，提高模型的鲁棒性，让模型可以持续地扩展深度，继续增加隐层。因此“局部感知、权值共享、下采样”被誉为卷积神经网络的三大核心思想，下面对卷积神经网络的三大核心思想进行详解。

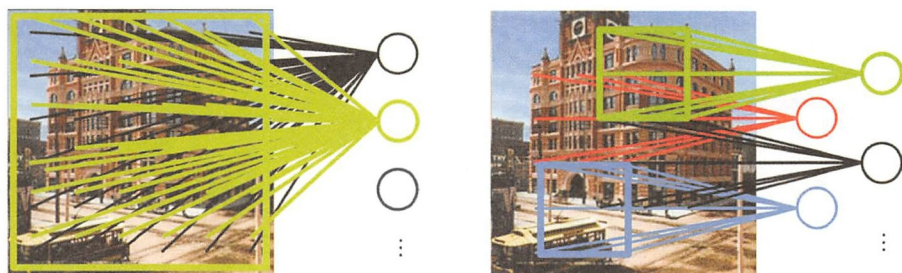
# 4.3.2 局部感知

针对图像中的空间关系，一般是局部的像素联系较为紧密，而距离较远的像素之间相关性则较弱。因此没有必要对全局图像进行感知，只需要对图像局部信息进行感知，然后在更深层网络中继续提取图像的局部信息。随着网络层次的深入，图像的尺寸逐渐缩小，对图像特征的提取也从片面到整体，综合起来便能够得到图像的全局信息。

图 4-14 (a) 为普通的神经网络全连接方式，全图像素（一个像素代表一个神经元节点）与下一层的所有神经元节点相连接，产生大量的连接。图 4-14 (b) 为卷积神经网络的局部连接方式，图像中局部区域的像素点只与下一层的某神经元节点相



连接，该局部区域的空间大小称为感知区域（Receptive Field），该感知区域实际上就是卷积核的空间大小。从图中可以看出，每个隐层神经元节点只负责连接到图像某个局部区域，从而大大减少网络中的权值参数。



(a) 全连接神经网络

(b) 局部连接神经网络

图 4-14 全连接与局部连接的区别。(a) 为对图像进行全连接操作，图像中的每一个像素都作为输入，连接到下一层的所有神经元上。(b) 为卷积神经网络中对图像进行局部连接操作，部分像素连接到下一层的一个神经元上

假设输入为  $1000 \times 1000$  大小的图像，按照一个像素值代表一个神经元节点，那么输入的神经元节点有  $1000 \times 1000 = 10^6$  个。继续假设下一层的隐层神经元节点数有 1000 个，从输入层到隐层，全连接的权重参数有  $1000 \times 1000 \times 1000 = 10^9$  个。从输入层到隐层，两层神经网络进行全连接产生上亿个权重参数。网络层数每增加一层，参数会呈指数式增长，当神经网络隐层到达 3 或 4 层时，会因为内存无法存下如此庞大的参数量而导致系统资源加载失败。

同样假设输入  $1000 \times 1000$  大小的图像，设定感知区域大小为  $10 \times 10$ （即该感知区域由  $10 \times 10$  个神经元节点组成的卷积核），隐层的神经元节点同样为 1000 个。那么经过局部感知后，每个感知区域对应一个隐层神经元节点，产生  $10 \times 10 \times 1000 = 10^5$  个权重参数。因此局部连接的方式参数数量是全连接的方式参数数量的千分之一。

由于每一层的输入图像和卷积核大小都不一样，因此会产生不同的感知区域，向下扩展网络的深度。另外，不同的感知区域能够感知图像中不同的纹理特征，从而随着卷积网络层的增加而获得更高维的图像特征。

### 4.3.3 权值共享

如果说一个卷积核在图像的一小块区域可以得到一个特定的纹理特征，那么在该图像其他有类似特征的地方，同样也可以使用该卷积核。

如图 4-15 所示，为了找到图像建筑物上的特征，卷积核 A 和卷积核 B 需要分别

检测各自滑动窗口位置上的特征。假设卷积核 A、B 两个滑动窗口都有着相同的纹理特征，那么实际上只需要一个卷积核 C 则可以代替卷积核 A 和 B 检测两个滑动窗口的特征。此时，可以共享卷积核 C（也就是共享相同的权值矩阵），把卷积核 C 学习到的特征作为探测器，应用到输入图像的其他地方，从而减少神经网络中的参数（减少重复的卷积核），该操作就是权值共享。

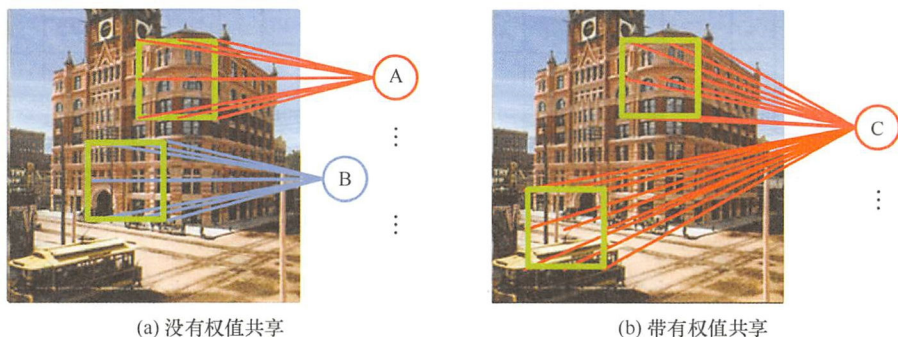


图 4-15 带有权值共享和没有权值共享的比较。(a) 为没有加入权值共享，每个滑动窗口中的所有像素值单独作为一个神经元的输入；(b) 为加入了权值共享，两个滑动窗口中的像素值可以作为同一个神经元的输入，这里对神经元进行了共享

假设该输入为  $1000 \times 1000$  的图像，每个神经元只与  $10 \times 10$  个像素值连接，由于引入了权值共享，从而减少隐层的神经元节点数（如设定下一层隐层神经元为 100 个而不是 1000 个），权值共享后卷积神经网络所需权重参数降为  $10 \times 10 \times 100 = 10^4$  个。减少卷积核的数量，实际上等同于减少网络中的权重参数，另外可以让监测到的数据之间的联系更加紧密，减少冗余检测信息。

### 4.3.4 下采样

在卷积神经网络中，输入给卷积层的图像可能很大，实际上并没有必要对原图进行操作，可以采用下采样 Pooling 技术，对输入的图像进行压缩，减少输出的总像素。如图 4-16 所示，每个卷积层后都带有一个 Pooling 层，通过下采样操作来缩减图像的空间尺寸规模。

Pooling 技术带来的好处如下：

- 减少过度拟合的可能性，当网络中权重参数过多时，很容易在训练阶段造成过度拟合；
- 缩减图像尺寸，减少计算量，提升计算速度；
- 进一步提取图像高维的统计特征。

常用的 Pooling 方法有如下两种。

- (1) 最大池化 (Max Pooling): 取 Pooling 窗口的最大值作为 Pooling 特征。
- (2) 均值池化 (Mean Pooling): 取 Pooling 窗口的均值作为 Pooling 特征。

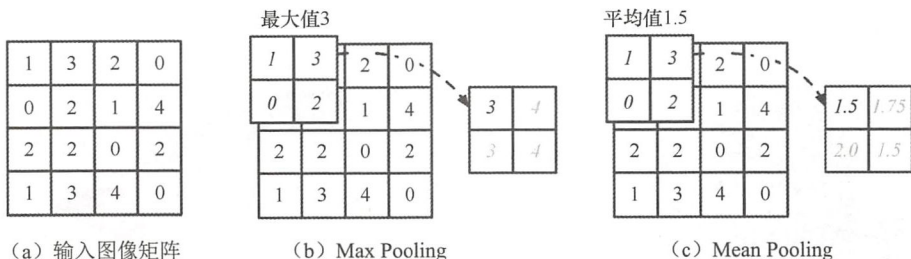


图 4-16 下采样 Pooling 操作示例, 窗口移动的步长设为 2 ( $stride=2$ )。(a) 图为一张  $4 \times 4$  大小的图像矩阵; (b) 图为 Max Pooling 操作, 滑动窗口为  $2 \times 2$  大小矩阵; (c) 图为 Mean Pooling 操作, 滑动窗口同样为  $2 \times 2$  大小矩阵

【代码清单 4-2】为 Max Pooling 算法实现。在实现算法之前我们需要知道: Pooling 窗口的大小、该 Pooling 窗口的滑动步长、输入矩阵。接着通过 Pooling 窗口大小和滑动步长, 确定输出矩阵的大小。最后把 Pooling 窗口的最大值按照滑动次序填入输出矩阵当中。

在 `max_pool_forward` 函数中, 首先分别获得 `pool_param` 字典里的参数和输入矩阵 `x` 的尺寸。计算出当前 Pooling 窗口在输入图像中按照步长 `stride` 能够向右和向下滑动的次数 `H_prime` 和 `W_prime`, 同时 `H_prime` 和 `W_prime` 也可以认为是 Pooling 层输出图像的高和宽。

接下来在 `for` 循环中, 计算出 Pooling 窗口左上角第一个点 (`h1, w1`) 到右下角最后一个点 (`h2, w2`) 的位置, 得到 Pooling 的窗口 `window`, 然后通过 `np.max()` 函数计算出 Pooling 窗口里的最大值, 记录到输出矩阵对应位置。

#### 【代码清单 4-2】Max Pooling 算法实现

```
def max_pool_forward(x, pool_param):
    (N, C, H, W) = x.shape          # 获取输入矩阵的大小
    height = pool_param['height']    # Pooling 窗口高度
    width = pool_param['width']      # Pooling 窗口宽度
    stride = pool_param['stride']    # Pooling 窗口滑动步长

    H_prime = 1 + (H - height) / stride  # 向下滑动的次数, 也为 pooling 输出的高度
    W_prime = 1 + (W - width) / stride    # 向右滑动的次数, 也为 pooling 输出的宽度

    out = np.zeros((N, C, H_prime, W_prime))  # 定义输出矩阵
```



```

# 遍历 batch
for n in xrange(N):
    for h in xrange(H_prime):
        for w in xrange(W_prime):
            h1 = h * stride                # (h1, w1) 为 pooling 窗口左上角第一个点
            w1 = w * stride
            h2 = h * stride + pool_height
            w2 = w * stride + pool_width    # (h2, w2) 为 pooling 窗口最后一个点
            window = x[n, :, h1:h2, w1:w2] # 获得当前 pooling 窗口
            win_1 = window.reshape((C, pool_height * pool_width))
            out[n, :, h, w] = np.max(win_1, axis=1)
return out

```

假设输入 Pooling 层的数据为一个  $4 \times 4$  的矩阵  $X$ ， $X$  实际上是一个  $(N, C, H, W)$  维矩阵。其中， $N$  为 batch 大小， $C$  为通道数， $H$  和  $W$  分别是图像的长度和宽度，设置  $X$  大小为  $[1, 1, 4, 4]$ ，表示  $X$  为 1 个 batch、1 个通道、 $4 \times 4$  大小图像的矩阵。Pooling 窗口大小为  $2 \times 2$ ，滑动步长为 2。

【代码清单 4-3】给出上述假设作为 Max Pooling 操作的输入，并给出显示结果。

【代码清单 4-3】执行 Max Pooling 算法并显示结果

```

>>> np.random.seed(8)
>>> x = np.random.randint(5, size=(1, 1, 4, 4)) # 随机产生一个 [1, 1, 4, 4] 的矩阵
>>> pool_param = {'height': 2, 'width': 2, 'stride': 2}
[[[ [1 1 2 3]
     [2 1 0 3]
     [4 0 4 1]
     [3 3 2 3]]]]

>>> out = max_pool_forward(x, pool_param)
[[[ [2.  3.]
     [ 4.  4.] ]]]

```

## 4.4 设计卷积神经网络架构

学习了卷积神经网络的三大思想和通过矩阵操作快速进行卷积计算后，我们或许可以结合神经网络的基本知识尝试去编写一个卷积神经网络模型，从而更好地理解其运作方式。在互联网科技巨头开源大量深度学习框架的时代，当务之急是设计出高效、可复用、精度高、损失小的卷积神经网络模型。



回顾近年来卷积神经网络的快速发展，从 AlexNet 网络模型开始，到 GoogleNet、VGGNet、ResNet 等优雅的卷积神经网络模型的出现，模型的运算时间不断减少、精度逐步提升、权重参数越来越少、网络深度动辄 20、30 层甚至更高，部分是基于网络层的不同排列方式和组织方式进行改造，部分是对网络层进行数学改造。如果我们也想要设计出一个惊艳的卷积神经网络，需要在理解的基础上进行实践与调参，下面来学习一些设计卷积神经网络架构的规律与技巧。

### 4.4.1 网络层间排列规律

卷积神经网络中最常见的是卷积层后接 Pooling 层，目的是减少下一次卷积输入图像大小，然后重复该过程，提出图像中的高维特征，最后通过全连接层得到输出。因此最常见的卷积神经网络结构如下：

$$\text{INPUT} \rightarrow [\text{CONV}] \rightarrow [\text{POOL}] \rightarrow [\text{FC}] \rightarrow \text{OUTPUT} \quad (4-4)$$

这里 CONV 为卷积层，POOL 为 Pooling 层，FC 为全连接层，[x] 为重复 x 层多次（例如 [CONV] 表示重复 CONV 卷积层多次）。常见的卷积神经网络结构规律如下。

- INPUT  $\rightarrow$  FC：实现一个简单的线性分类器。
- INPUT  $\rightarrow$  CONV：实现一个滤波操作，如高斯模糊、中值滤波等。
- INPUT  $\rightarrow [\text{CONV} \rightarrow \text{POOL}] \times 2 \rightarrow \text{FC} \rightarrow \text{OUTPUT}$ ：经过两次卷积层接 Pooling 层，实现小规模卷积神经分类网络。
- INPUT  $\rightarrow [\text{CONV} \rightarrow \text{CONV} \rightarrow \text{POOL}] \times 3 \rightarrow [\text{FC}] \times 2 \rightarrow \text{OUTPUT}$ ：多次连续两个卷积层后接一个 Pooling 层，该思路适用于深层次网络（如 VGGNet、ResNet 等）。因为在执行 Pooling 操作前，多次小卷积核卷积可以有效减少网络权重参数，从输入数据中学习 to 更高维特征。

### 4.4.2 网络参数设计规律

卷积神经网络模型中的参数定义比人工神经网络模型要复杂，需要注意的地方也不一样，卷积神经网络模型超参数设计的一般规律如下。

- 输入层矩阵的大小应该可以被 2 整除多次。常用数字包括 32、64、96、224、384 或者 512。其中 224 是 AlexNet 的经典输入大小，该要求是为了方便网络的卷积层和 Pooling 层计算，如每次 Pooling 层产生的特征图是输入的一半，尽量减少在数学约减时造成的数据丢失。
- 卷积层尽量使用小尺寸卷积核。网络层数越深，卷积核尺寸应该设置得越小。从空间上来说，随着网络层次加深，其输出特征图越小，相对来说，

意味着卷积核越大（感知区域越大）。为了避免空间上卷积核增大而导致特征图减小，且图像中的感知区域太大难以提取输入数据的高维特征，应尽量使用小尺寸卷积核（如  $1 \times 1$ 、 $3 \times 3$ 、 $5 \times 5$ ）。另外，在性能方面，卷积核越小，卷积神经网络所需权重参数越少，可以有效地加快运算速度。

- **卷积步长尽量不要过大。**在实际应用中，更小的步长提取特征效果更好，如设置步长为 1 可以让空间维度的下采样操作由 Pooling 层负责，卷积层只负责对输入数据进行特征提取。
- **卷积层中应使用 Same Padding 零填充矩阵边界。**使用 Same Padding 零填充边界可以让卷积层的输出数据保持和输入数据大小不变。如果卷积层只进行卷积而不进行零填充，那么数据体的尺寸就会略微减小，随着网络层次的加深，图像边缘的信息就会过快地损失掉。
- **Padding 的设置与卷积核大小有关。**当卷积核大小  $K=3$ （即卷积矩阵大小为  $3 \times 3$ ），则设置  $padding=1$  来保持卷积操作中的输出尺寸不变性；当  $K=5$ ，设置  $padding=2$ 。对于任意卷积核大小  $K$ ， $padding=(K-1)/2$  能保持尺寸不变性。
- **Pooling 层一般使用  $2 \times 2$  窗口，步长为 2 的 Max Pooling 操作。**Pooling 层负责对输入数据在空间维度进行压缩（下采样），其中 Max Pooling 的 Pooling 窗口尺寸很少会大于 3，当下采样操作过于激烈时，容易造成数据信息丢失，导致卷积神经网络性能急剧下降。
- **全连接层数不宜超过 3 层（ $FC \leq 3$ ）。**全连接层数越多，训练难度越大，容易造成过拟合和梯度消散，因此一般卷积神经网络最后接的是大部分为两个连续的全连接层加一个输出分类层。

## 4.5 示例1: 可视化手写字体网络特征

1989 年（Yann LeCun et al, 1989）发表的 LeNet5 网络架构揭开了深度学习的神秘面纱，从此深度学习开始被人们所熟知。LeNet5 网络架构的成功应用，不断地推动着深度学习领域的发展。

本节将会介绍卷积神经网络模型中最为经典的 LeNet5 模型，该模型是第一个成熟的卷积神经网络模型，几乎所有能够搜索到的卷积神经网络资料都会提及该网络模型结构。有了 LeNet5 模型和 MNIST 手写字体数据库后，我们会对网络模型中产生的特征张量进行可视化分析，让读者对卷积神经网络模型的运作方式有初步的认知。同时也希望读者明白卷积神经网络模型并不是一个看不透的黑盒子，而是一个潘多拉之盒，等待着我们去窥探其中的精彩！

## 4.5.1 MNIST手写字体数据库

MNIST 数据集是一个手写体数据集（如图 4-17 所示），数据集中每一个样本都是 0~9 的手写数字，该数据集由 4 部分组成：训练图片集、训练标签集、测试图片集和测试标签集。其中，训练集中有 60000 个样本，测试集中有 10000 个样本，每张照片均由  $28 \times 28$  大小的灰度图像组成。为了便于存储和下载，官方对 MNIST 数据集的图片进行集中处理，将每一张图片拉伸成为 (1, 784) 的向量表示。



图 4-17 MNIST 手写字体数据集

首先根据数据库官方给出的参数设置手写字体数据图片的大小和分类大小，在【代码清单 4-4】中使用 `mnist.load_data()` 函数从 Amazon 服务器中下载数据集，并把数据集分为训练图片集、训练标签集、测试图片集和测试标签集。在读入训练图片集和测试图片集后，还会对图片集进行  $x/255$  的计算操作，目的是将图片像素归一化到 [0, 1] 区间内，方便将数据输入卷积神经网络模型中进行计算。

### 【代码清单 4-4】加载 MNIST 手写字体数据库

```
img_rows, img_cols = (28, 28) # 设置单张图像大小
num_classes = 10             # 设置分类大小

def get_mnist_data():
    """
    加载 mnist 手写字体数据集
    """
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    # 对数据进行 reshape
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
```



```

# 图像数据归一化
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

# 获得所属标签
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

return x_train, y_train, x_test, y_test

>>> x_train, y_train, x_test, y_test = get_mnist_data()

```

## 4.5.2 LeNet5网络模型

LeNet5 网络模型是为了识别手写字体和计算机打印字符而设计，该模型在手写识别领域非常成功，曾被广泛应用于美国银行支票手写体识别。

LeNet5 网络模型作为卷积神经网络中的开创性工作，提取了三大思想：

- 局部感知；
- 下采样；
- 权值共享。

因为图像特征分布在图像的像素上，利用卷积操作可以在多个位置提取相类似的特征，于是有了局部感知。另外由于当年并没有计算能力强悍的 GPU 来辅助训练神经网络，因此通过下采样层有效地加快训练和提取更高维特征，能够节省参数和计算，这与当年的技术相比是一个关键的优势。另外原论文中提到，全卷积不应该被放在第一层，图像特征有着高度的空间相关性，因此权值共享可以充分利用图像上的空间相关性。

下面为 LeNet5 网络模型架构详情。LeNet5 共有 7 层（不包含输入），每层都包含可训练参数，其网络模型架构如图 4-18 所示，输入为一张手写字体的图像，经过卷积层→ Pooling 层→卷积层→ Pooling 层，后接两个全连接层，最后输出为输入图像所属的分类。

- 输入层：LeNet5 网络的输入为  $32 \times 32$  大小的图像，比 MNIST 数据集的图片稍微大一些。
- C1 层（卷积层）：使用 6 个大小为  $5 \times 5$  的卷积核对输入图像进行卷积操作，卷积后得到的特征图尺寸为 28，因此产生 6 个大小为  $28 \times 28$  的特征图。



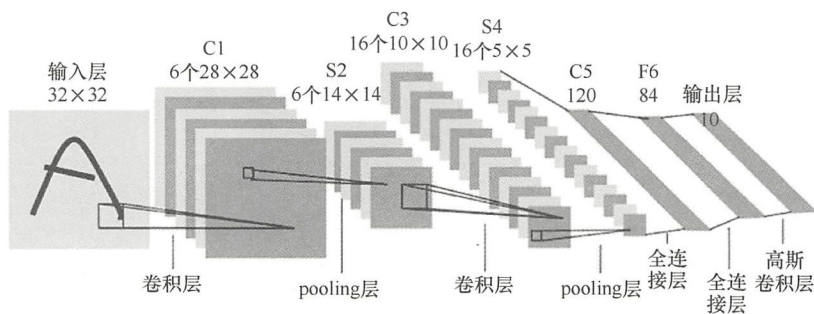


图 4-18 LeNet5 网络模型架构图

- S2 层（Pooling 层）：这里采用 Max Pooling 操作，Pooling 窗口大小为  $2 \times 2$ 。经过 Pooling 层后得到 6 个  $14 \times 14$  的特征图，作为下一层的输入。
- C3 层（卷积层）：使用 16 个大小为  $5 \times 5$  的卷积核对输入的特征图进行卷积操作。值得注意的是，C3 中输出的特征图是 S2 中的特征图进行加权组合得到的。输出为 16 个  $10 \times 10$  的特征图。
- S4 层（Pooling 层）：同样采用 Max Pooling 操作，Pooling 窗口大小为  $2 \times 2$ 。最后输出 16 个  $5 \times 5$  的特征图，神经元个数为 400 ( $16 \times 5 \times 5$ )。
- C5 层（全连接层）：该层可以理解对 S4 层产生的特征向量进行拉伸，每一个像素代表一个神经元，使用全连接操作输出特征为 120 个神经元。同样可以当作使用 120 个大小为  $5 \times 5$  的卷积核对输入的特征图进行卷积操作（没有进行 Padding 操作），这样 C5 层的输出 120 个特征图大小变为  $5-5+1=1$ 。
- F6 层：该层与 C5 层进行全连接，输出特征为 84 个神经元。
- 输出层：该层与 F6 层全连接，输出长度为 10，代表所抽取的特征属于哪个类别。例如特征向量  $[0,0,0,0,0,1,0,0,0,0]$ ，1 在索引位置 5，故该特征向量表示输入的图片属于第 5 个分类。

LeNet5 网络模型定义如【代码清单 4-5】所示，输入使用 MNIST 数据集中  $28 \times 28$  大小的手写字体图像，第一个卷积层使用 32 个大小为  $3 \times 3$  的卷积核，第二个卷积层使用 64 个大小为  $5 \times 5$  的卷积核，并在网络中加入 Dropout 层用于防止过度拟合。

#### 【代码清单 4-5】定义 LeNet5 网络模型

```
def LeNet5(w_path=None):

    input_shape = (img_rows, img_cols, 1)
    img_input = Input(shape=input_shape)
```

```

x = Conv2D(32, (3, 3), activation="relu", padding="same", name="conv1")(img_input)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool1')(x)
x = Conv2D(64, (5, 5), activation="relu", name='conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='pool2')(x)
x = Dropout(0.25)(x)

x = Flatten(name='flatten')(x)

x = Dense(120, activation='relu', name='fc1')(x)
x = Dropout(0.5)(x)
x = Dense(84, activation='relu', name='fc2')(x)
x = Dropout(0.5)(x)
x = Dense(10, activation='softmax', name='predictions')(x)

model = Model(img_input, x, name='LeNet5')
if(w_path): model.load_weights(w_path)

return model

```

最后 LeNet5 网络模型显示如【代码清单 4-6】所示。

#### 【代码清单 4-6】LeNet5 网络模型

```

>>> lenet5 = LeNet5()
>>> lenet5.summary()

```

Model loaded.

Layer (type)	Output Shape	Param #
=====		
input_4 (InputLayer)	(None, 28, 28, 1)	0
conv1 (Conv2D)	(None, 28, 28, 32)	320
pool1 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2 (Conv2D)	(None, 10, 10, 64)	51264
pool2 (MaxPooling2D)	(None, 5, 5, 64)	0
dropout_7 (Dropout)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
fc1 (Dense)	(None, 120)	192120

dropout_8 (Dropout)	(None, 120)	0
fc2 (Dense)	(None, 84)	10164
dropout_9 (Dropout)	(None, 84)	0
predictions (Dense)	(None, 10)	850
=====		
Total params: 254,718		
Trainable params: 254,718		
Non-trainable params: 0		

### 4.5.3 LeNet5网络训练

在网络训练的过程中，需要保存网络训练的结果参数文件，这里使用了 Keras 的 ModelCheckpoint 函数来保存网络在迭代训练过程中每一层产生的模型参数。训练损失函数使用交叉熵损失函数，优化器使用 Adadelta 算法（随机梯度下降算法的变种），迭代次数选择 30 次，具体如【代码清单 4-7】所示。

【代码清单 4-7】LeNet5 网络模型训练过程

```
from keras.callbacks import ModelCheckpoint

# 检查保存断点文件夹是否存在
if not os.path.exists('lenet5_checkpoints'):
    os.mkdir('lenet5_checkpoints')

# 编译 LeNet5 网络
lenet5.compile(loss=keras.losses.categorical_crossentropy,
               optimizer=keras.optimizers.Adadelta(),
               metrics=['accuracy'])

checkpoint = ModelCheckpoint(monitor='val_acc',
                             filepath = 'lenet5_checkpoints/model_{epoch:02d}_{val_acc:.3f}.h5',
                             save_best_only = True)

# 开始 LeNet5 训练网络
lenet5.fit(x_train, y_train,
          batch_size = 128,
          epochs = 30,
          verbose = 1,
```

```

validation_data = (x_test, y_test),
callbacks = [checkpoint])

# 验证训练 LeNet5 网络的结果
score = lenet5.evaluate(x_test, y_test, verbose = 0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

```

最终 LeNet5 网络训练结果如【代码清单 4-8】所示，中间隐藏了部分迭代结果，网络的训练集为 60000 张图像，验证集为 10000 张图像，最终的精度为 99.53%。

#### 【代码清单 4-8】训练结果

```

Train on 60000 samples, validate on 10000 samples

Epoch 1/30
60000/60000 [=====] - 6s 95us/step - loss:0.0618 - acc:0.9832 - val_loss:0.0299 - val_
acc:0.9915
Epoch 2/30
60000/60000 [=====] - 5s 88us/step - loss:0.0591 - acc:0.9844 - val_loss:0.0272 - val_
acc:0.9920
Epoch 3/30
60000/60000 [=====] - 5s 88us/step - loss:0.0535 - acc:0.9851 - val_loss:0.0258 - val_
acc:0.9923
Epoch 4/30
60000/60000 [=====] - 5s 88us/step - loss:0.0527 - acc:0.9858 - val_loss:0.0243 - val_
acc:0.9929
Epoch 5/30
60000/60000 [=====] - 5s 87us/step - loss:0.0524 - acc:0.9862 - val_loss:0.0242 - val_
acc:0.9929
.....

Epoch 26/30
60000/60000 [=====] - 5s 86us/step - loss:0.0315 - acc:0.9914 - val_loss:0.0210 - val_
acc:0.9945
Epoch 27/30
60000/60000 [=====] - 5s 86us/step - loss:0.0312 - acc:0.9914 - val_loss:0.0212 - val_
acc:0.9944
Epoch 28/30
60000/60000 [=====] - 5s 87us/step - loss:0.0320 - acc:0.9912 - val_loss:0.0231 - val_
acc:0.9933
Epoch 29/30
60000/60000 [=====] - 5s 86us/step - loss:0.0310 - acc:0.9919 - val_loss:0.0190 - val_
acc:0.9948
Epoch 30/30

```



```
60000/60000 [=====] - 5s 88us/step - loss:0.0322 - acc:0.9918 - val_loss:0.0197 - val_
acc:0.9953
```

```
Test loss:0.0197388240156
```

```
Test accuracy: 0.9953
```

从上面的代码中可以看到，最终预测精确率和测试精确率均已经达到 99% 以上，在【代码清单 4-9】使用 sklearn 中的分类报告函数 `classification_report` 来显示该网络模型对手写字体识别的准确率表格（如图 4-19 所示）。

	precision	recall	f1-score	support
0	0.07	0.02	0.03	980
1	0.00	0.00	0.00	1135
2	0.11	0.93	0.20	1032
3	0.02	0.02	0.02	1010
4	0.76	0.01	0.03	982
5	0.00	0.00	0.00	892
6	0.00	0.00	0.00	958
7	0.00	0.00	0.00	1028
8	0.00	0.00	0.00	974
9	0.21	0.08	0.11	1009
avg / total	0.12	0.11	0.04	10000

图 4-19 LeNet5 手写字体识别的分类参数报告

#### 【代码清单 4-9】训练结果

```
from sklearn.metrics import classification_report

y_pred = lenet5.predict(x_test)
y_pred = np.argmax(y_pred, axis = 1)
y_test = np.argmax(y_test, axis = 1)
print(classification_report(y_test, y_pred))
```

## 4.5.4 可视化特征向量

使用 LeNet5 网络模型训练完 MNIST 手写字体数据集后，细心的读者可能会很好奇卷积神经网络对输入的图像执行了什么操作？输入的图像经过卷积神经网络每一层会产生什么样的特征图？

本节我们使用【代码清单 4-10】随机在测试集中选择了一张图像（如图 4-20 所示），作为 LeNet5 网络模型的输入，并对网络模型的每一层特征进行可视化分析。

#### 【代码清单 4-10】测试结果与预测结果比较

```
# 随机在测试集中选出一张手写字体
>>> i = np.random.choice(x_test.shape[0])
```

```
>>> plt.imshow(x_test[i, 0], interpolation='None', cmap='gray')

>>> print("{} label:{}".format(i, y_train[i,:]))
>>> 1144 label:[ 0.  0.  0.  0.  0.  1.  0.  0.  0.]
```

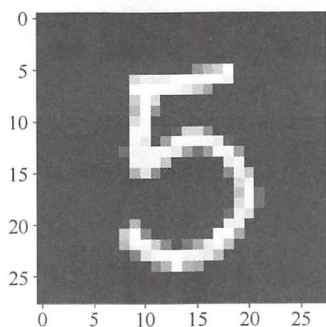


图 4-20 测试集第 1144 张图像，代表数字“5”

得到了输入图像之后，接下来继续对 LeNet5 网络每一层产生的特征进行可视化分析操作。在【代码清单 4-11】中，分为 `get_activations` 和 `display_activations` 两个函数，前者用于获得网络模型每一层的激活值（特征），后者用于把激活值（特征）进行转换数据排列格式便于图像化显示。

经过卷积层和 Pooling 层操作之后产生高维特征图（特征张量）。例如，经过第一个卷积层后产生特征张量为  $(1, 28, 28, 32)$ ，我们需要把特征转换为  $(28, 28 \times 32)$  大小的图像进行可视化操作，于是需要使用 `numpy` 的 `transpose` 改变原特征的排列方式，然后通过 `numpy` 的 `hstack` 合并特征，最终效果如图 4-21 所示。

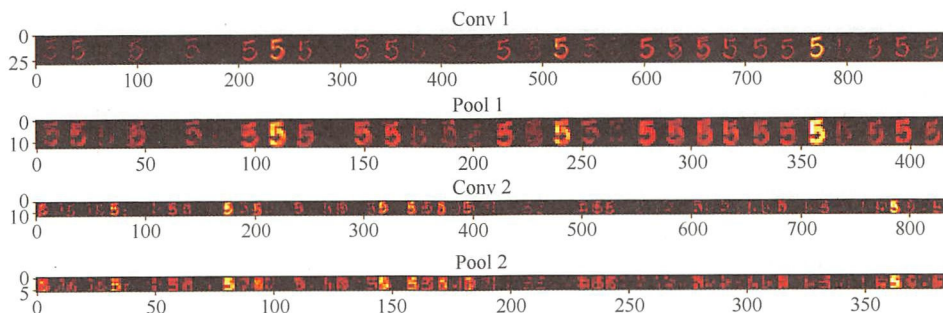


图 4-21 可视化卷积层和 Pooling 层产生的特征张量。颜色越浅，表示数值越高；颜色越深，表示数值越低

在全连接层和激活层产生的特征向量的可视化操作中，产生的特征为二维特征，对其特征经过 `numpy` 的 `repeat` 复制特征多次，并叠加在 `axis=0` 上。例如经过第

一次全连接层后输出特征为 (1, 120)，经过上述复制特征向量操作后特征变为 (10, 120)，再进行可视化输出，最终效果如图 4-22 所示。

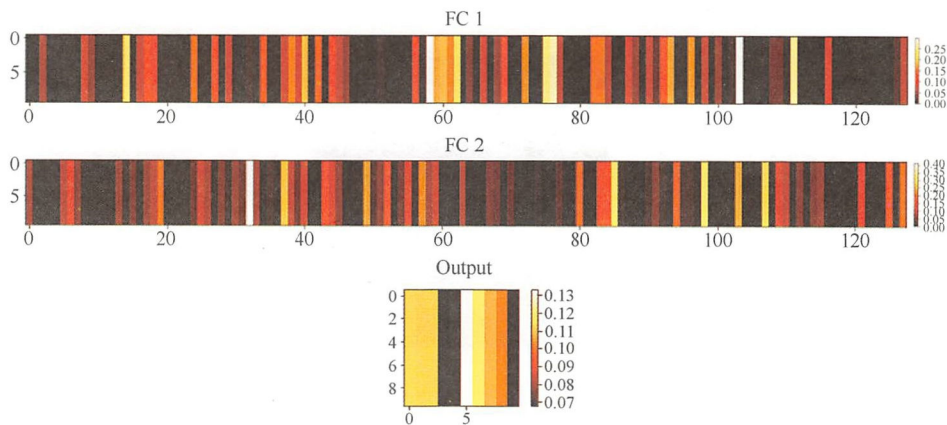


图 4-22 可视化全连接层和输出层的特征向量。颜色越浅，表示数值越高；颜色越深，表示数值越低

#### 【代码清单 4-11】 可视化卷积层的特征图

```
def get_activations(model, model_input, layer_name = None):
    """
    获得网络模型的每一层的激活值
    """
    activations = []
    inp = [model.input]

    # CNN 模型的所有层输出
    model_layers = [layer.output for layer in model.layers if
                     layer.name == layer_name or layer_name is None]

    # 模型所在层对应的激活值函数输出
    funcs = [K.function(inp + [K.learning_phase()], [layer]) for layer in model_layers]
    list_inputs = model_input.reshape(1,1,28,28)

    # 获得每一层的输出激活值并存储在 activations 列表中
    layer_outputs = [func([list_inputs]) for func in funcs]
    for activation in layer_outputs:
        activations.append(activation)

    return activations

def display_activations(activations_tensor):
```

```

"""
网络模型激活值图形显示化
"""

for i, activation_map in enumerate(activations_tensor):
    activation_map = activation_map[0]
    shape = activation_map.shape
    # 卷积层或者 Pooling 层产生的特征张量
    if len(shape) == 4:
        tt = np.hstack(np.transpose(activation_map[0], (0, 1, 2)))
        plt.imshow(tt, interpolation='None', cmap='hot')
        plt.show()
    # 全连接和输出层产生的特征向量
    if len(shape) == 2:
        tt = np.repeat(matrix, 10, axis=0)
        plt.imshow(tt, interpolation='None', cmap='hot')
        plt.colorbar()
        plt.show()

>>> activations = get_activations(lenet5, x_test[i])
>>> display_activations(activations)

```

LeNet5 网络模型小结如下。

- LeNet5 使用卷积神经网络中的 3 层架构：卷积、下采样、非线性激活函数。
- 使用卷积提取图像空间特征。
- 下采样利用了图像平均稀疏性。
- 激活函数采用了 Tanh 或 Sigmoid 函数。
- 多层神经网络（MLP）作为最后的分类器。
- 各层之间使用稀疏连接矩阵，以避免高计算成本。

总的来说，LeNet5 作为深度学习用于图像分类任务的具有影响力的架构，其中有很多思想值得我们去学习与借鉴，近几年来神经网络架构大多数是基于 LeNet5 的三大特性进行展开和改进的。

## 4.6 示例2: 少样本卷积神经网络分类

2012 年 Alex Krizhevsky 发表的 AlexNet 夺冠 ImageNet 图像分类大赛后，AlexNet 网络模型成为了近年来最经典的卷积神经网络模型，可以毫不夸张地说，AlexNet 的成功掀起了一场深度学习的视觉革命。此外，随着 GPU 性能越来越强大，卷积神经



网络模型训练时间不断减少，并且能够处理分辨率越来越大的图像和数亿万计的权重参数，因此卷积神经网络当之无愧地成为了深度学习的主力军。

有一种说法是“深度学习训练需要拥有海量的数据才会有意义”，因为深度学习是从数据中自动地学习特征，没有足够的样本，就无法学习足够多的特征，尤其当输入的数据维度很高时。我在刚开始学习卷积神经网络时，如果卷积神经网络模型精度无法继续提高，那么我首先做的就是增加样本。这种做法不完全正确，卷积神经网络作为深度学习的重要支柱，被认为是针对“感知”问题最好的模型之一，即使只有少量数据作为输入，其网络模型也能够学习到数据中较为准确的高维特征。实际上，即使输入小数据集，卷积神经网络依然能够得到较为合理的结果，并不需要任何特征工程作为辅助。

本例中将会使用 AlexNet 网络模型对少量的数据集进行训练，并使用该网络模型对数据进行预测分类。在本例的最后，我们会针对过度拟合问题对卷积神经网络模型进行微调（fine-tune），以期减少过度拟合的情况。

## 4.6.1 Kaggle猫狗数据库

本章将会以 Kaggle 猫狗大战中的数据集作为出发点，使用 AlexNet 对猫狗数据集进行预测。Kaggle 猫狗原数据集各有 12500 张猫的照片和狗的照片（如图 4-23 所示），本例中只取了各类别的前 1000 张图片用于 AlexNet 模型训练，另外从各类别中取 400 张图片作为测试集。

其中，猫狗数据集的文件结构存储方式如【代码清单 4-12】所示。

### 【代码清单 4-12】数据集文件结构存放方式

```
data/
  train/      # 训练集
    dogs/
      dog001.jpg, dog002.jpg, ..., dog1000.jpg
    cats/
      cat001.jpg, cat002.jpg, ..., cat1000.jpg
  test/       # 测试集
    dogs/
      dog001.jpg, dog002.jpg, ..., dog400.jpg
    cats/
      cat001.jpg, cat002.jpg, ..., cat400.jpg
```

在 2013 年之前，想对该猫狗数据集进行有效的分类是非常困难的。计算机领域的视觉科学家们认为，如果没有重大的突破，那么对于分类器来说，分类精度很难超过 60%。那么在猫狗数据集中使用少量的样本数据的卷积神经网络，可以获得超

过 80% 的分类精度吗？答案毋庸置疑是肯定的，这就是深度学习在感知领域取得重大突破的原因之一。



图 4-23 猫狗数据中示例。图中上半部分为“狗”图，下半部分为“猫”图

【代码清单 4-13】中首先定义好每一次训练的 batch 大小，不建议将 batch 设置得过大或过小。当 batch 过大时，梯度下降算法每次迭代的训练时间会增加；当 batch 过小时，每次训练的数据量过少，导致算法难以收敛（详见第 2、3 章相关内容）。

接下来训练集和测试集分别使用了 ImageDataGenerator 对象。对于训练集 train dataset 来说，需要进行归一化，通过旋转、放大、翻转等操作增加训练数据集中的样本，对训练集数据进行扩展能够有效地增加识别准确率。另外，对于测试集 test dataset 只进行归一化 rescale 操作，方便卷积神经网络进行数值计算。

通过 flow\_from\_directory 函数读取目录中的图片数据，产生用于训练和分类的数据和标签。

#### 【代码清单 4-13】数据多样化

```
batch_size = 16 # 每批次训练集的大小

train_datagen = ImageDataGenerator(
    rescale=1./255, shear_range=0.2,
    zoom_range=0.2, horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    'CatDogData/train',
    shuffle=True, # 打乱数据集中的顺序
    target_size=input_shape[1:],
    batch_size=batch_size,
    class_mode='categorical')
```

```
validation_generator = test_datagen.flow_from_directory(  
    'CatDogData/test',  
    shuffle=True, # 打乱数据集中的顺序  
    target_size=input_shape[1:],  
    batch_size=batch_size,  
    class_mode='categorical')
```

在实际工程当中，分类数量可能会比该例子更复杂，例如分类有“野马”“斑马”“河马”等，也可能如 ImageNet 数据集有 1000 种类别。卷积神经网络的神奇之处在于不用担心分类类别多、每个分类的图像数据量庞大、单个分类物体的多样性等问题，因为卷积神经网络在 GPU 的帮助下能够处理大数据问题，高层模型可以有效提取高维特征，进行多分类识别。最后我们需要做的是认真检查所定义的卷积神经网络模型，告诉卷积神经网络最后的预测类别后即可开始有趣的训练过程。

## 4.6.2 AlexNet模型

上面提到 AlexNet 网络模型的成功，掀起了一场深度学习的视觉革命。AlexNet 取得如此辉煌成绩的原因主要有以下 4 点。

(1) 使用 **ReLU** 函数作为激活函数：降低了 Sigmoid 激活函数的计算量，并且有效地避免了过度拟合和梯度消散等问题，提高了卷积神经网络的预测准确率。

(2) 使用 **Dropout** 技术：在训练期间选择性地去掉不重要的神经元，避免卷积神经网络模型过度拟合。

(3) 引入 **Max Pooling** 下采样操作：减少网络参数的同时，提高预测准确率。

(4) 利用双 **GPU** 架构：大大减少了模型训练时间。

图 4-24 所示为 AlexNet 的网络模型，其网络架构一共有 5 个卷积层、3 个 max pooling 层、2 个全连接层（值得注意的是三、四卷积层后没有 Pooling 层），其输入矩阵为  $3 \times 224 \times 224$  大小的三通道彩色图像。

下面为 AlexNet 网络模型架构详情。

- 输入层 (Input)：输入为  $3 \times 224 \times 224$  大小图像矩阵。
- 卷积层 (Conv1)：96 个  $11 \times 11$  大小的卷积核（每个 GPU 上 48 个卷积核）。
- Pooling 层 (Pool1)：Max Pooling 窗口大小为  $2 \times 2$ ， $stride=2$ 。
- 卷积层 (Conv2)：256 个  $5 \times 5$  大小的卷积核（每个 GPU 上 128 个卷积核）。
- Pooling 层 (Pool2)：Max Pooling 窗口大小为  $2 \times 2$ ， $stride=2$ 。
- 卷积层 (Conv3)：384 个  $3 \times 3$  大小的卷积核（每个 GPU 上各 192 个卷积核）。
- 卷积层 (Conv4)：384 个  $3 \times 3$  大小的卷积核（每个 GPU 上各 192 个卷积核）。
- 卷积层 (Conv5)：256 个  $3 \times 3$  大小的卷积核（每个 GPU 上各 128 个卷积核）。

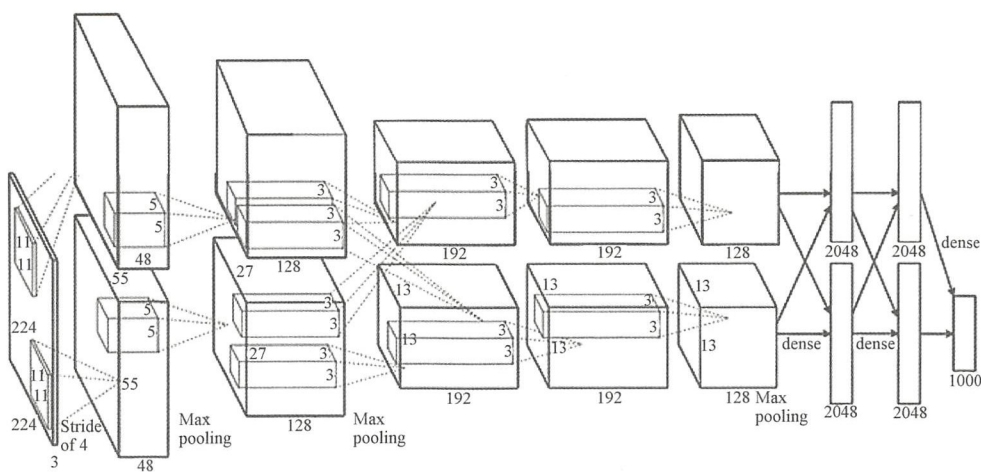


图 4-24 使用了双 GPU 架构的 AlexNet 网络模型结构。其卷积核大小从左到右分别为：11、5、3、3、3，输入的图像大小为  $224 \times 224$ ，最后一次 Pooling 后图像大小为  $13 \times 13$

- Pooling 层 (Pool5)：Max Pooling 窗口大小为  $2 \times 2$ ， $stride=2$ 。
- 全连接层 (FC1)：第一个全连接层，将第五层 Pooling 层的输出连接成为一个一维向量作为该层的输入，输出 4096 个神经节点。
- 全连接层 (FC2)：第二个全连接层，输入输出均为 4096 个神经节点。
- Softmax 输出层：输出为 1000 个神经节点，输出的每个神经节点单独对应图像所属分类的概率。因为在 ImageNet 数据集中有 1000 个分类，因此设定输出维度为 1000。

【代码清单 4-14】中使用 Keras 来实现 AlexNet 模型，这里并没有完全按照图 4-24 的 AlexNet 模型设置双 GPU，只是简单地按照 AlexNet 模型定义参数进行设置。代码最后使用对象 Model 来实例化 AlexNet 网络模型。

为了方便全连接进行计算，代码中进入第六层 (dense1) 前把卷积层得到的特征矩阵进行 Flatten 操作，改变上一层输出的矩阵形式，把三维的矩阵数据一维向量化。此操作常用于卷积层到全连接层的过渡衔接。

#### 【代码清单 4-14】AlexNet 网络模型架构实现

```
from keras.models import Model
from keras.layers import Flatten, Dense, Input
from keras.layers import Convolution2D, MaxPooling2D
from keras.preprocessing import image

nb_classes = 2 # 需要预测的分类
input_shape=(3, 227, 227) # 输入的图像大小为三通道 227×227
```



```
# Input Layer 输入层
inputs = Input(shape=input_shape, name="input")

# Layer1 第一层: 两个卷积操作和两个 Pooling 操作
conv1_1 = Conv2D(48, (11, 11), strides=(4, 4), activation="relu", name="conv1_1")(inputs)
conv1_2 = Conv2D(48, (11, 11), strides=(4, 4), activation="relu", name="conv1_2")(inputs)
pool1_1 = MaxPooling2D((3, 3), strides=(2, 2), name='pool1_1')(conv1_1)
pool1_2 = MaxPooling2D((3, 3), strides=(2, 2), name='pool1_2')(conv1_2)

# Layer2 第二层: 对第一层产生的作为两个卷积层和 Pooling 层的独立输入
conv2_1 = Conv2D(128, (5, 5), activation="relu", name="conv2_1", padding="same")(pool1_1)
conv2_2 = Conv2D(128, (5, 5), activation="relu", name="conv2_2", padding="same")(pool1_2)
pool2_1 = MaxPooling2D((3, 3), strides=(2, 2), name='pool2_1')(conv2_1)
pool2_2 = MaxPooling2D((3, 3), strides=(2, 2), name='pool2_2')(conv2_2)

# Merge 合并层: 第二层进入第三层的时候把数据混合合并
merge1 = concatenate([pool2_2, pool2_1], axis=1)

# Layer3 第三层: 分别进行两个卷积操作
conv3_1 = Conv2D(192, (3, 3), activation="relu", name="conv3_1", padding="same")(merge1)
conv3_2 = Conv2D(192, (3, 3), activation="relu", name="conv3_2", padding="same")(merge1)

# Layer4 第四层: 与第三层一样分别进行两个卷积操作
conv4_1 = Conv2D(192, (3, 3), activation="relu", name="conv4_1", padding="same")(conv3_1)
conv4_2 = Conv2D(192, (3, 3), activation="relu", name="conv4_2", padding="same")(conv3_2)

# Layer5 第五层: 分别对第四层的数据进行卷积操作和 Pooling 操作
conv5_1 = Conv2D(128, (3, 3), activation="relu", name="conv5_1", padding="same")(conv4_1)
conv5_2 = Conv2D(128, (3, 3), activation="relu", name="conv5_2", padding="same")(conv4_2)
pool5_1 = MaxPooling2D((3, 3), strides=(2, 2), name='pool5_1')(conv5_1)
pool5_2 = MaxPooling2D((3, 3), strides=(2, 2), name='pool5_2')(conv5_2)

# Merge 合并层: 第五层进入全连接层之前需要把分开的合并
merge2 = concatenate([pool5_1, pool5_2], axis=1)
# 通过 Flatten 把多维的输入一维化
dense1 = Flatten(name='flatten')(merge2)

# Layer6、Layer7 第六层、第七层: 进行两次 4096 维的全连接, 中间加入 Dropout 层避免拟合
dense2 = Dense(4096, activation='relu', name='dense2')(dense1)
dense2 = Dropout(0.5)(dense2)
dense3 = Dense(4096, activation='relu', name='dense3')(dense2)
dense3 = Dropout(0.5)(dense3)
```

```
# Output Layer 输出层: 输出层输出 nb_classes 个类别, 输出分类函数使用 softmax
dense_3 = Dense(nb_classes, name='dense_3')(dense3)
prediction = Activation("softmax", name="softmax")(dense_3)

# 最后定义模型的输入与输出
AlexNet = Model(input=inputs, outputs=prediction)
```

定义完 AlexNet 网络模型后, 可以使用对象 AlexNet 的 summary 函数来查看模型的信息, 从【代码清单 4-15】中可以看出, 最后该 AlexNet 模型使用的参数大约为  $5.6 \times 10^7$  个。实际上卷积神经网络的模型参数是非常庞大的, 如果没有三大核心思想对网络模型的参数进行裁剪,  $3 \times 227 \times 227$  大小的图像在 AlexNet 网络中产生的权重参数规模将会达十亿级, 现在只需要千万级别的参数即可。

【代码清单 4-15】查看代码清单 4-14 中定义的 AlexNet 网络模型信息

```
>>> AlexNet.summary()
```

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	(None, 224, 224, 3)	0
conv1 (Conv2D)	(None, 56, 56, 96)	34944
pool1 (MaxPooling2D)	(None, 27, 27, 96)	0
conv2 (Conv2D)	(None, 27, 27, 256)	614656
pool2 (MaxPooling2D)	(None, 13, 13, 256)	0
conv3 (Conv2D)	(None, 13, 13, 384)	885120
conv4 (Conv2D)	(None, 13, 13, 384)	1327488
conv5 (Conv2D)	(None, 13, 13, 256)	884992
pool5 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten (Flatten)	(None, 4096)	0
fc1 (Dense)	(None, 4096)	16781312
fc2 (Dense)	(None, 4096)	16781312

```

predictions (Dense)          (None, 1)          4097
=====
Total params: 37,313,921.0
Trainable params: 37,313,921.0
Non-trainable params: 0.0
-----
None

```

### 4.6.3 AlexNet训练

定义完卷积神经网络模型并且准备好数据后，需要做的就是编译和训练该 AlexNet 网络。

在 AlexNet 网络模型编译前，我们需要定好损失函数和模型优化器。如【代码清单 4-16】所示，损失函数 loss 使用了交叉熵代价函数（categorical\_crossentropy），metrics 参数为每次迭代 epoch 之后是否显示预测精度和测试集的精度。优化器使用随机梯度下降方法（SGD），lr 为学习率，decay 为衰减系数，momentum 为动量。

训练阶段的参数中（fit\_generator）总迭代次数设置为 80 次，单次迭代中训练集为 16 张图片。

#### 【代码清单 4-16】编译和训练 AlexNet 模型

```

# CNN 网络模型优化算法使用随机梯度下降
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
alexnet.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

# 训练模型，训练结果保存在 history_callback
history_callback = alexnet.fit_generator(
    train_generator,
    steps_per_epoch=2000 # batch_size,
    epochs=80,
    validation_data=validation_generator,
    validation_steps=800 #batch_size
)

# 训练完存储训练预测的结果与模型中的权重参数
pandas.DataFrame(history_callback.history).to_csv("./AlexNet_model.csv")
alexnet.save_weights('./AlexNet_model.h5')

```

训练完 AlexNet 网络后，使用 pandas 保存训练的结果，save\_weights 保存模型的权重参数，最后的训练结果如【代码清单 4-17】所示。

**【代码清单 4-17】 AlexNet 模型对猫狗数据集训练结果**

```

Epoch 1/80
62/62 [=====] - 26s - loss:0.6959 - acc:0.5106 - val_loss:0.6924 - val_
acc:0.5104
Epoch 2/80
62/62 [=====] - 23s - loss:0.6907 - acc:0.5182 - val_loss:0.6866 - val_
acc:0.5026
Epoch 3/80
62/62 [=====] - 25s - loss:0.6899 - acc:0.5413 - val_loss:0.6828 - val_
acc:0.5130
...
...
Epoch 79/80
62/62 [=====] - 22s - loss:0.0991 - acc:0.9678 - val_loss:0.4683 - val_
acc:0.8464
Epoch 80/80
62/62 [=====] - 22s - loss:0.0786 - acc:0.9733 - val_loss:0.6544 - val_
acc:0.7865

```

为了能够更加方便地观察数据,【代码清单 4-18】中使用 `plot_performance` 函数来对保存训练结果的精度、损失进行绘图。另外,如果 Keras 使用 TensorFlow 作为底层接口,在训练阶段还可用 TensorFlow 自带的可视化工具 `tensorBoard` 来实时查看训练效果,或者使用 NVIDIA 自带的可视化工具 NVIDIA Digital 来显示实时训练效果。

**【代码清单 4-18】显示函数 `plot_performance` 实现**

```

def plot_performance(history):
    ''' 显示训练集与测试集的精确率和损失曲线 '''
    plt.subplot(1,2,1)
    plt.plot(history['acc'][1:])
    plt.plot(history['val_acc'][1:], 'r')
    plt.title('Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['train', 'val'], loc='upper left')

    plt.subplot(1,2,2)
    plt.plot(history['loss'][1:])
    plt.plot(history['val_loss'][1:], 'r')
    plt.title('Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['train', 'val'], loc='upper left')

```



```

plt.tight_layout()
plt.show()

>>> # 训练结果保存的数据顺序
>>> column_names = ['NaN', 'acc', 'loss', 'val_acc', 'val_loss']

>>> # 读取训练结果
>>> df = pandas.read_csv('./AlexNet_model.csv', header=None, names=column_names)

>>> # 显示效果图
>>> plot_performance(df)

```

本例中使用 Kaggle 猫狗数据集的 AlexNet 模型在 40 次迭代后，训练集和测试集的准确率在 78% ~ 82%，继续往下迭代则出现了过度拟合情况，测试集的精确度不再提升，损失值也开始波动（如图 4-25 所示）。虽然最后测试集的精确率没有达到 90% 以上，但是本例中只采用了原数据集中约 8% 的数据，也没有对模型和超参数进行优化等系列操作。最终在测试集中能够得到 80% 左右的精确率，本次 AlexNet 神经网络的训练结果是较为令人满意的。

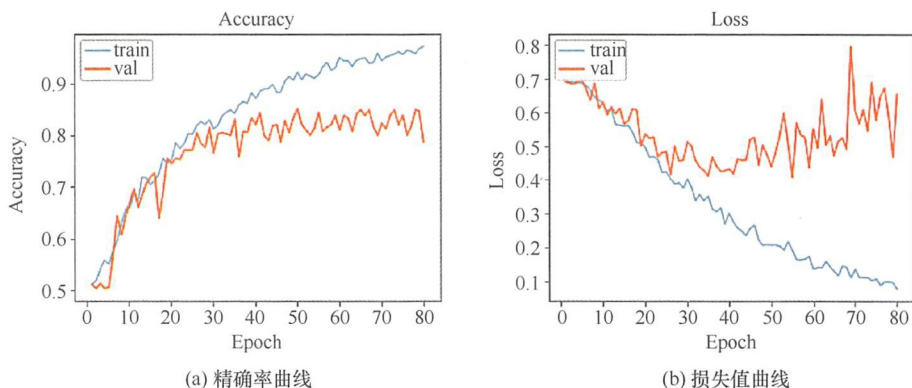


图 4-25 (a) 为训练与测试集的精确率曲线，(b) 为训练与测试集的损失曲线。经过 80 次迭代后，训练集的精度不断攀升，最终达到 98% 左右，而测试集的精度最初与测试集曲线类似，当迭代次数超过 40 后，准确率在 80% 左右来回波动；损失值训练集的损失值一直减少到 0.1，而测试集的损失值从最初的减少，到 50 次迭代后损失值来回波动

## 4.6.4 AlexNet 预测

将训练后的网络权重参数保存为 AlexNet\_model.h5。在预测阶段需要加载该网络权重参数文件，然后输入单张图片，利用该网络对新输入的数据进行预测，并给出

预测分类结果。

【代码清单 4-19】中得到的是一维向量，该向量为各分类的预测概率（独立同分布）。通过对 `preds` 向量使用 `numpy.argsort()[-1]` 获取概率最大的下标，然后迭代求分类标签 `predict_idx` 中所属的分类。

从代码运行结果中可以看到，最终输入的图片预测的结果与真实的结果一致。

【代码清单 4-19】加载 AlexNet 网络并进行预测

```
>>> from keras.preprocessing import image

# 查看分类标签的索引
>>> predict_idx = validation_generator.class_indices
{'cat': 0, 'dog': 1}

# 读取网络权重参数文件
>>> model.load_weights('./AlexNet_model.h5', by_name=True)

# 读入待预测的图片，并把图片转换为输入的格式
>>> img_path = './CatDogData/validation/dog/dog1000.jpg'
>>> img = image.load_img(img_path, target_size = input_shape[1:])
>>> x = image.img_to_array(img)
>>> x = np.expand_dims(x, axis = 0) # (1, 3, 227, 227) x 矩阵大小
>>> x = x.reshape((-1, ) + input_shape) / 255. # 对应归一化 rescale=1./255 操作

# 向前传播得到预测值
>>> preds = model.predict(x)
[[ 0.18342575  0.81657422]]

# 根据分类标签输出预测所属分类
>>> top_indices = pred.argsort()[-1:][::-1]
>>> result = [key for key, value in predict_idx.items() if(value == top_indices)]
>>> print("Predict result is: {}".format(result))

Predict result is: ['dog']
```

## 4.6.5 微调网络

从图 4-19 的 AlexNet 模型训练效果曲线图中可知，该卷积神经网络训练精确率比测试精确率高 10% 左右，这一现象很大程度上是由过度拟合造成的。本例中仅使用样本的 8% 数据用于训练，所以应该对过度拟合的问题多加注意。为了检验该 AlexNet 模型是否真正过度拟合，或者想进一步提高网络的预测精度，可以对该

AlexNet 模型进行微调 (Fine-tune)。

在微调之前,我们将训练集中部分图片和测试集中的图片进行对调,以加强输入数据的多元性。本次微调的思路仿照论文 Full Training or Fine Tuning (Nima Tajbakhsh et al., 2016) 的内容,其基本思路是分层训练。假设有五层的卷积神经网络模型 [L1, L2, L3, L4, L5],在第一轮迭代时,只对 L5 层的权重参数进行微调,冻结 L1 ~ L4 层。在下一轮迭代时,对 L4、L5 层进行训练微调,冻结 L1 ~ L3 层,依次类推。从本质上来说,微调的训练是从深层网络渗透到浅层网络中去的。

【代码清单 4-20】为微调的参数设置, layers 为需要进行微调的网络层, epochs 是每层迭代次数, lr 是每次迭代对应的学习率,学习率随着网络迭代次数的增加而减少。

#### 【代码清单 4-20】神经网络微调参数设置

```
# 定义需要重新训练的网络层
layers = ['dense_3', 'dense3', 'dense2', 'pool5_1', 'conv4_1',
          'concatenate_1', 'conv2_1', 'conv1_1']

# 对应重新训练的网络层迭代次数
epochs = [10, 10, 10, 10, 10, 10, 10, 10]

# 对应重新训练网络层随机梯度下降的学习率
lr = [1e-2, 1e-3, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4, 1e-4]
```

【代码清单 4-21】为对 AlexNet 网络模型的微调代码,首先我们需要迭代定义好需要调整参数的网络层。遇到允许训练的网络层,则设置该层 trainable 参数为 True;否则冻结该层,不允许在训练阶段让优化器更新该层网络的权重参数。然后执行训练函数,开始微调训练该 AlexNet 网络模型。

#### 【代码清单 4-21】微调 AlexNet 网络

```
# 开始迭代需要重新训练的网络层
for i, layer in enumerate(layers):
    # 标记指定 layer 是为可训练的层
    for layer in model.layers:
        if layer.name == layer:
            layer.trainable = True
        layer.trainable = False # 其余的层冻结,不能够训练

    # 编译该层网络的权重参数
    sgd = SGD(lr=lr[i], decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=sgd,
                  metrics=['accuracy'])

# 训练该层的网络参数,每层训练只需要迭代 epoch(10) 次
```

```

for epoch in range(epochs[i]):
    history = model.fit_generator(
        train_generator,
        steps_per_epoch=2000 // batch_size,
        epochs=1,
        validation_data=validation_generator,
        validation_steps=800 // batch_size
    )

# 存储 fine-tune 后的权重参数
model.save_weights('./AlexNet_model_2.h5')

```

网络经过微调后的精确率曲线如图 4-26 所示, 具体数值见【代码清单 4-22】。从精确率曲线图中可以看出, 该网络的训练集和预测集的平均准确率在 80% 上下波动 2 个百分点, 预测与测试的精度之间的差别也控制在 2% 以内, 因此可以说明该 AlexNet 模型并没有过度拟合。

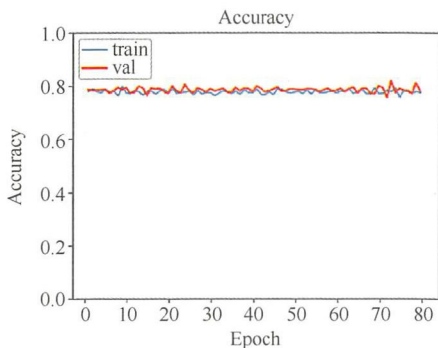
#### 【代码清单 4-22】 AlexNet 网络微调输出

```

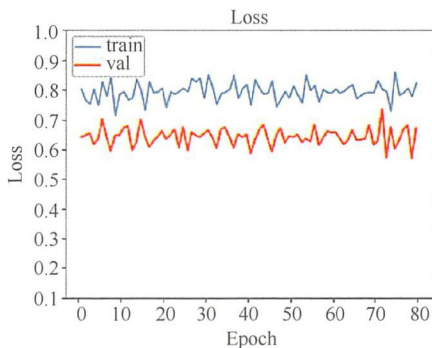
Epoch 1/1
125/125 [===] - 32s - loss:0.8032 - acc:0.7815 - val_loss:0.6417 - val_acc:0.7875
Epoch 1/1
125/125 [===] - 31s - loss:0.7668 - acc:0.7910 - val_loss:0.6470 - val_acc:0.7850
...
Epoch 1/1
125/125 [===] - 24s - loss:0.7781 - acc:0.7815 - val_loss:0.5723 - val_acc:0.8137
Epoch 1/1
125/125 [===] - 25s - loss:0.8239 - acc:0.7770 - val_loss:0.6718 - val_acc:0.7825

plot_performance(history_finetune)

```



(a) 精确率曲线



(b) 损失值曲线

图 4-26 微调后的测试集与训练集的曲线



读者可能会有疑问，为什么经过网络微调后没有提升该网络分类精确率呢？

实际上我们只是把训练集和测试集中的数据部分进行了对调，并修改了学习率。这一改变对于神经网络的微调是微不足道的，而本例的意图是让读者明白如何通过修改代码进行卷积神经网络的微调训练。如果想要通过微调网络获得更好的结果，可以尝试如下更多的方法：

- 加入更多的数据和增加数据的多元性；
- 增加 Dropout 层，加大 Dropout 的概率；
- 使用 L1 或 L2 正则项；
- 微调更多的卷积层；
- 输入更好的初始化值。

当上述的方法不能够提高 AlexNet 网络的预测精度时，一种更好的选择是调整模型的网络参数，即调整网络模型的层间结构和卷积参数，也可以尝试使用 VGG-16、VGG-19、GoogleNet、ResNet 等先进的网络模型代替 AlexNet 等。

由于深度学习模型具有很强的可复用性，所以非常方便对网络进行微调。例如，把一个在大规模数据上训练好的网络模型重用在外另外的数据集上，只需要对部分网络层进行微调训练，即可达到一定的效果。

互联网上不同的深度学习软件框架都会发布许多免费下载的预训练模型，方便我们进行重用，以提升在小数据集上的性能（对于网络微调的更多技巧可以参考第 3 章内容）。

## 4.7 本章小结

卷积神经网络在深度学习中占据了举足轻重的地位，本章详细介绍了卷积神经网络的网络结构，通过了解卷积神经网络在实际工程案例中的应用，使读者明确卷积神经网络能够出色地处理图像任务和一些特定的非图像任务。为了更加深入地了解卷积神经网络，我们探索了其三大核心思想：局部感知、权值共享、下采样技术。既然是卷积神经网络，就不得不提卷积操作。从图像滑动窗口进行卷积操作到网络卷积层的操作，再到使用矩阵的方式快速地进行卷积计算，本章全面讲解了卷积神经网络中的卷积计算。在进一步学习卷积神经网络之前，还需要简单介绍它的网络架构的层间排列规律和参数设计规律，以便后续更好地设计属于我们自己的网络模型。

（1）卷积神经网络主要由输入层、卷积层、Pooling 层、全连接层、输出层组成，其中卷积层用于提取图像的高维特征，Pooling 层用于下采样。

(2) 传统神经网络用于图像处理的缺点是：网络参数量庞大、容易造成相邻像素间信息丢失，严重制约网络模型的深度发展。

(3) CNN 的三大核心思想：

- 局部感知，神经元间进行局部连接从而减少网络参数，并能够提取网络高维特征；
- 权值共享，共享卷积核减少冗余特征，减少参数的同时提高检测准确率；
- 下采样，提取重要特征，减少网络参数。

(4) Pooling 操作：分为 Max Pooling 和 Mean Pooling。Max Pooling 取窗口内的最大像素值，Mean Pooling 取窗口内平均像素值，一般 Max Pooling 比 Mean Pooling 更能够有效地提取图像中的重要特征。

(5) Padding 操作：为了控制卷积后生成的特征图大小，一般对输入矩阵图像的外边缘进行零值填充。默认使用 Same Padding 操作。

(6) 使用矩阵进行快速卷积的操作步骤：使用 Im2col 算法把输入图像和卷积核转换成为规定的矩阵排列方式，然后使用 GEMM 算法对转换后的两个矩阵相乘。

(7) 多层连续小卷积核的卷积层组合方式，能够更加有效地提取图像中的高维特征，并能显著地减少网络参数。

## 引用/参考

- [1] Russakovsky O, Li F F. Attribute Learning in Large-Scale Datasets[M]// Trends and Topics in Computer Vision. Springer Berlin Heidelberg, 2012:1-14.
- [2] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C]// International Conference on Neural Information Processing Systems. Curran Associates Inc. 2012:1097-1105.
- [3] Kim Y. Convolutional Neural Networks for Sentence Classification[J]. Eprint Arxiv, 2014.
- [4] Zhang R, Isola P, Efros A A. Colorful Image Colorization[J]. 2016:649-666.
- [5] Cheng Z, Yang Q, Sheng B. Deep Colorization[C]// IEEE International Conference on Computer Vision. IEEE Computer Society, 2015:415-423.
- [6] Larsson G, Maire M, Shakhnarovich G. Learning Representations for Automatic Colorization[M]// Computer Vision – ECCV 2016. Springer International Publishing, 2016:577-593.
- [7] Tajbakhsh N, Shin J Y, Gurudu S R, et al. Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?[J]. IEEE Trans Med Imaging, 2017, 35(5):1299-1312.
- [8] Funke C M, Gatys L A, Ecker A S, et al. Synthesising Dynamic Textures using Convolutional Neural Networks[J]. 2017.

- [9] Gatys L A, Ecker A S, Bethge M. Texture Synthesis Using Convolutional Neural Networks[J]. 2015, 70(1):262-270.
- [10] Gatys L A, Ecker A S, Bethge M. A Neural Algorithm of Artistic Style[J]. Computer Science, 2015.
- [11] Zhu C, Byrd R H, Lu P, et al. Algorithm 778 : L-BFGS-B : Fortran subroutines for large-scale bound-constrained optimization[J]. Acm Transactions on Mathematical Software, 1997, 23(4):550-560.
- [12] Johnson J, Alahi A, Li F F. Perceptual Losses for Real-Time Style Transfer and Super-Resolution[J]. 2016:694-711.
- [13] Yosinski J, Clune J, Nguyen A, et al. Understanding Neural Networks Through Deep Visualization[J]. Computer Science, 2015.
- [14] Poznanski A, Wolf L. CNN-N-Gram for Handwriting Word Recognition[C]// Computer Vision and Pattern Recognition. IEEE, 2016:2305-2314.
- [15] Kenshimov C, Bampis L, Amirgaliyev B, et al. Deep learning features exception for cross-season visual place recognition[J]. Pattern Recognition Letters, 2017.

---

# 第 5 章

---

## 卷积神经网络 视觉盛宴

本章主要内容：

- 卷积神经网络图像目标检测
- 卷积神经网络图像语义分割



目前，计算机视觉（Computer Vision）毫无疑问成为了深度学习应用领域中发展最为迅猛的方向之一。卷积神经网络模型在图像分类领域中的检测精度已经达到了与人类同等的水平，并开始探索计算机视觉领域的新问题——图像检测、图像语义分割、图像标注、图像生成、图像场景识别等。

在第4章中，我们介绍了卷积神经网络中两个经典的网络模型（LeNet 和 AlexNet），利用卷积神经网络模型可以提取输入图像的高维特征。可是在现实工程当中，我们不满足于单纯的图像分类任务：当一张图像里有多个类别时，我们想要知道图像中到底有多少个分类；当一张图像中有多个物体且每个物体所在的位置都不一样时，我们想要获得每个物体在图像中的具体位置。面对上述任务，我们应该怎么去处理呢？

让人兴奋与激动的是，卷积神经网络模型可以解决上述问题。在本章中，首先会介绍近年来深度学习在图像检测方面的最新技术，从2s检测一张图像的R-CNN到超实时的YOLO网络模型，机器视觉与深度学习的算法相碰撞可以给我们带来更多的灵感。接着，介绍深度学习在图像语义分割方面的最新技术，从卷积神经网络CNN到全卷积网络FCN，彰显了其无限魅力。

在本章结束之前，我们还会使用Python实现YOLO网络模型中用到的非极大值抑制算法（NMS），从多个候选框中选出目标框。接着，实现候选框提取（Selective Search, SS）算法，其中会融合性地介绍与机器视觉相关的内容。这些例子会为我们踏进深度机器视觉领域打下坚实的基础。深度学习是计算机视觉天梯的垫脚石，让我们一起步入卷积神经网络的视觉盛宴，尽情地畅想机器视觉的未来！

## 5.1 图像目标检测

如图5-1所示，从人类视觉来说，我们可以清楚地看到左侧图片中的物体和它们所处的位置。从图像中找到3个物体分别是狗、人和马，其所在位置右侧图片的方框所示。给定一张图像，找出图像中所有目标的位置，并给出每个目标的具体所属类别，这一过程便是图像目标检测。

得益于丰富的视觉神经和大脑神经，目标检测对人类而言可以说是一项非常简单的任务。可是从计算层面来说，输入的是一个数值为0~255的多维矩阵，在没有其他传感器的辅助下，我们很难去对多维矩阵的图像中的目标物体进行层次和结构的划分，特别是受到光照、物体旋转平移的影响，目标物体表面的纹理特征开始改变，这些因素都严重制约着机器视觉的发展。

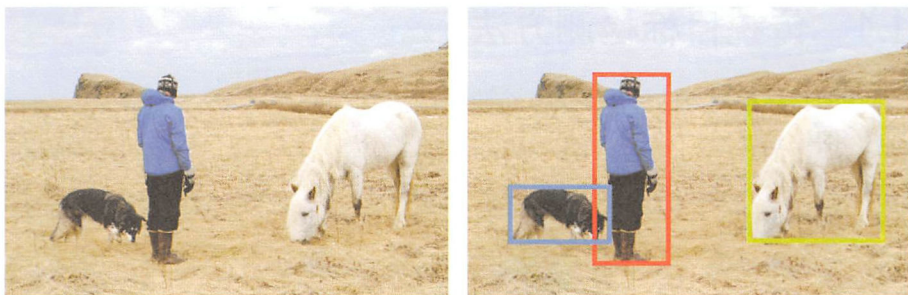


图 5-1 图像目标检测。左为原图，右图中方框为经过图像目标检测算法检测后得到的图像，并识别出方框内物体的类别

近年来，深度学习中的卷积神经网络在为图像分类任务带来革命性的变革后，又迎来了另外一个更加有挑战的任务——图像检测。从 2014 年开始，图像目标检测取得巨大的突破，学界先后涌现出 R-CNN、Fast R-CNN、Faster R-CNN 等结合候选框建议（Region Proposal）的网络框架，以及超实时的 YOLO、SSD 等基于回归的目标检测框架。

#### 候选区域建议（Region Proposal, RP）

候选区域建议很好地解决了在图像中滑动窗口的缺点，通过算法预先找到图像中目标可能出现的位置框。

其基本原理是利用图像的纹理、边缘、颜色等特征信息，保证在选取有限个窗口（几百到几千个窗口不等）的前提下，尽可能提高找到目标窗口的概率。这极大地降低了后续操作的时间复杂度，且提升了系统的稳定性。常用的候选区域建议算法有选择性搜索（Selective Search, SS）和 Edge Boxes，本章的例子中将会实现选择性搜索（SS）算法。

深度学习在图像检测领域的检测精度足以与人类的双眼相媲美。PASCAL VOC 数据集上目标检测平均精度（mean Average Precision, mAP）从 R-CNN 的 53.3% 发展到 Faster R-CNN 的 75.9%，而今最新的检测精度已经超过了 85%。其检测速度也得到进一步突破，最初 R-CNN 一帧图像需要数秒，到 Faster R-CNN 的 200ms 左右一帧，再到 YOLO 的超实时 200fps 一帧，深度学习让图像检测真正走进了实际工程项目当中。

本节我们会深入了解：

- （1）传统目标检测网络框架流程；
- （2）以 R-CNN 为代表的结合卷积神经网络和候选区域建议算法的目标检测框架；
- （3）以 YOLO 为代表的把目标检测问题转换为机器学习中回归问题的目标检测框架；
- （4）提高目标检测性能的技巧和方法。

## 5.1.1 传统目标检测方法

传统的目标检测方法一般分为4个阶段：

- 图像预处理；
- 目标区域选择；
- 特征提取；
- 分类器分类。

对于一张输入图像首先会对其进行降噪、平滑等预处理工作，然后在给定图像上选择一些目标出现概率较高的候选区域，接着对这些候选区域进行特征值提取，最后使用分类器对提取到的特征值进行分类，得到候选框所属的类别。

### 1. 图像预处理

图像预处理的主要目的是消除与检测目标无关的信息，恢复图像中有用的真实信息，增强有关信息的可检测性并最大限度地简化数据，从而改进特征抽取、图像分割、匹配和识别的可靠性。常用的方法有高斯滤波、均值滤波、图像腐蚀和膨胀、二值化等。

### 2. 目标区域选择

对目标物体可能出现的位置进行定位。由于目标可能出现在图像中的任何位置，且目标的大小、长宽比例在一开始可能无法确定，因此最原始的方法是采用不同尺寸大小的滑动窗口对全图进行遍历。

### 3. 特征提取

对图形中目标区域的窗口进行特征提取，常用的图像特征有颜色特征、纹理特征、形状特征、空间关系特征等。这个阶段是目标检测中最为重要的阶段，因为所提取的特征好坏直接影响最后分类结果的准确率。

对图形中目标区域的窗口进行特征提取，常用的图像特征有颜色特征、纹理特征、形状特征、空间关系特征等。这个阶段是目标检测中最为重要的阶段，因为所提取的特征好坏程度直接影响最后分类结果的准确率。

### 4. 分类器分类

将特征提取的结果表示成向量形式，交给特征分类器进行分类，给出所属分类的概率（这里使用到的分类方法属于有监督学习，因此需要预先对人工标注的特征进行训练）。

可是传统目标检测方法主要有两个问题。

- 滑动窗口：基于滑动窗口的区域选择策略没有针对性，时间和空间复杂度高，产生大量无用的特征。
- 特征工程：需要人工对目标区域选择合适的特征，工程时间长。另外，经过特征工程选择的特征不一定能够符合物体多样性特征，其鲁棒性较差。



由于传统的目标检测方法存在着诸多不足，于是机器视觉相关的研究者开始思考：既然卷积神经网络能够高效地提取图像物体的高维特征，那么为什么不结合深度神经网络来实现图像目标检测呢？

## 5.1.2 基于区域的网络

### 1. R-CNN 网络

2014 年，(Ross B. Girshick et al.) 使用候选区域建议算法结合卷积神经网络设计了 R-CNN 网络框架。使用深度学习的方法代替了传统的目标检测方法（滑动窗口和特征工程结合），在目标检测方向取得巨大突破，并开启了基于深度学习目标检测的热潮。

图 5-2 为 R-CNN 预测阶段的流程架构图，其网络框架流程如下。

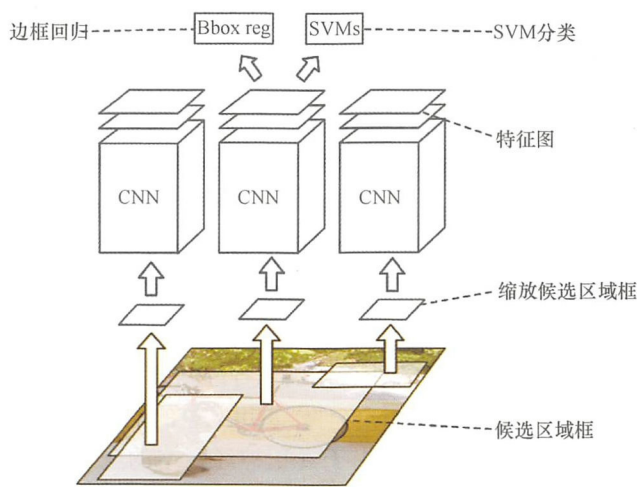


图 5-2 R-CNN 网络架构图。原图经过选择性搜索（SS）算法得到多个候选区域，候选区域经过卷积神经网络后使用 SVM 进行物体预测和 Bounding-box 回归

#### R-CNN网络框架流程

- (1) 原始图像经过选择性搜索（SS）算法提取出约 2000 个候选区域框。
- (2) 把所有候选框缩放到固定大小（如  $227 \times 227$ ）。
- (3) 候选框图像逐个通过 CNN 网络模型提取其特征。
- (4) 使用 SVM 分类器对 CNN 提取的特征进行分类。
- (5) 通过线性回归算法对候选框位置进行微调（边框回归）。



(1) 首先, 图像经过候选区域建议算法提取出约 2000 个候选区域框, 这 2000 个候选区域基本上包括了图像中可能出现的目标物体 (Region Proposal 阶段)。

(2) 为了统一卷积神经网络模型的全连接层, 需要确保输入特征数据的维度不变性, 我们将每一个候选区域缩放到同一尺寸, 如  $227 \times 227$  (Resize 阶段)。

(3) 接着将每个  $227 \times 227$  的候选区域输入卷积神经网络中, 最后获得 4096 维的特征向量 (神经网络倒数第 2 层)。

(4) 2000 个候选区域和其对应的卷积神经网络提取到的特征向量进行组合, 可以得到  $2000 \times 4096$  维的特征矩阵  $M$ 。在这之前我们已经使用大量 4096 维的特征向量进行训练, 得到  $k$  个 SVM 分类器 ( $k$  为分类数目)。接下来, 我们将该特征矩阵  $M$  与  $k$  个 SVM 分类器组成的权值矩阵  $N_{4096 \times k}$  相乘, 最终获得矩阵  $S_{2000 \times k}$  ( $M_{2000 \times 4096} \otimes N_{4096 \times k}$ )。其中,  $S$  矩阵中的元素  $s_{ij}$  表示第  $i$  个候选框属于第  $j$  个分类的概率。

(5) 分别对概率矩阵  $S$  中每一列 (每一类别) 进行非极大值抑制 NMS, 去除重叠候选区域, 得到该类别中得分高且重合度低的候选区域。最后还需要对 SVM 分好类的候选区域进行边框回归 (Bounding-box Regression)。

### R-CNN 小结

R-CNN 作为基于深度学习目标检测算法的鼻祖, 存在着以下不足。

- **重复计算量大, 速度慢:** 将每个候选框输入卷积神经网络中获得其单独特征, 2000 个候选窗口需要执行 2000 次卷积神经网络操作, 即使在使用 GPU 的前提下, R-CNN 网络框架处理一张图像仍然需要 40s 以上。
- **训练阶段步骤烦琐:** R-CNN 训练分为微调卷积神经网络、训练 SVM 分类器、训练边框回归器 3 个单独的步骤, 且每次训练都会产生一定的偏差。
- **训练耗时且占用空间大:** 由于使用卷积神经网络和 SVM 分类器单独训练, 因此需要保存大量的特征文件, 占用大量的硬盘空间。
- **候选框缩放丢失图像信息:** 由于卷积神经网络的全连接层神经元数固定, 因此需要将输入候选框的矩阵大小缩放到相同尺度, 该预处理操作会造成图像失真, 部分信息丢失。

#### 边框回归 (Bounding-box Regression)

边框回归是对候选框进行纠正微调的线性回归算法, 其目的是让候选区域窗口与真实窗口更加吻合, 最终得到每个类别经过回归算法修正后的 Bounding-box。

## 2. SPP-Net

R-CNN 存在的最大问题是:

- 大量重复的计算;

- 对候选框进行缩放，造成图像信息丢失。

SPP-Net 的出现就是为了解决上述两个问题：不管输入的图像尺寸大小如何，都能够正确地传入卷积神经网络模型中，并且只执行一次卷积神经网络操作就能获得所有的候选区域框的特征。

实际上，2000 个候选区域框属于图像的一部分，因此我们可以对图像进行一次卷积神经网络特征提取，然后将原图上的候选区域框位置映射到最后一层卷积后的特征图上，接着在对应映射位置上进行分类和回归处理，这样就不需要为每一个候选区域框都执行一次卷积神经网络分类，极大地减轻了网络架构的计算量。

图 5-3 所示为 SPP-Net 预测阶段的流程架构图，该网络框架流程如下。

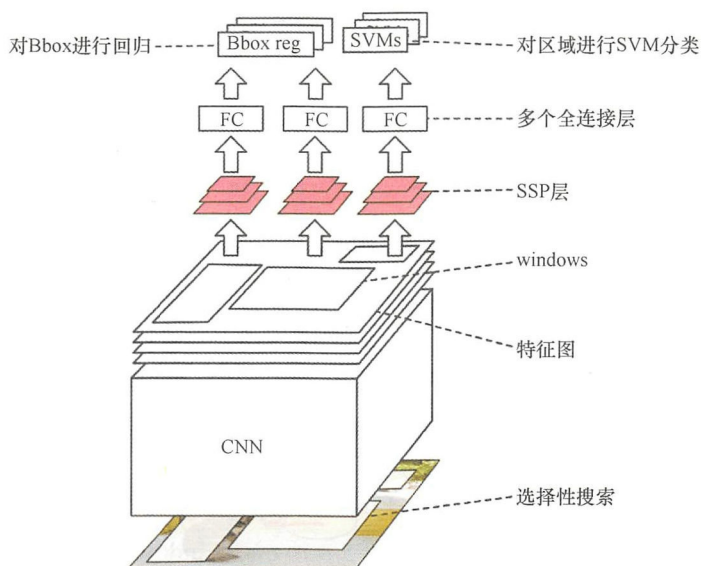


图 5-3 SPP-Net 架构图。原图经过卷积神经网络和选择性搜索 (SS) 算法得到特征图，每个 window 经过 SSP 层后使用 SVM 进行预测和 Bounding-box 回归

### SPP-Net 的网络框架流程

- (1) 原始图像经过选择性搜索 (SS) 算法提取出约 2000 个候选区域框；
- (2) 原始图像通过卷积神经网络网络直到最后一层卷积层，获得其特征图；
- (3) 对选择性搜索 (SS) 算法提取到的候选框映射在特征图，称为特征图上的 window；
- (4) 对每个 window 进行 SPP 操作，获得固定的特征向量传入后续的全连接层；
- (5) 对全连接层输出的特征向量使用 SVM 进行分类，边框回归算法微调 Bounding-box 的位置与大小。

(1) 首先, 与 R-CNN 一样, 将图像经过选择性搜索 (SS) 算法提取出约 2000 个候选区域框, 这 2000 个候选区域基本上包括了图像中可能出现的目标物体 (Region Proposal 阶段)。

(2) 将原始图像作为卷积神经网络模型的输入, 经过网络模型最后一层卷积层, 得到  $n$  个特征图 (假设基础卷积神经网络为 AlexNet, 则最后一层卷积层后有 256 个特征图)。

(3) 将步骤 (1) 中基于原始输入图像找到的约 2000 个候选区域位置映射到最后一层卷积层得到特征图, 映射后的窗口称为 windows。假设原图有 2000 个候选区域, 最后一层卷积层产生 256 张特征图, 通过映射, 一共可以得到  $2000 \times 256$  个 windows。

(4) 由于全连接层的输入为固定长度的特征, 但每个候选区域的大小不一, 因此映射后的 windows 也都大小不一, 直接把 windows 对应的特征作为全连接层的输入是不可能的。而 SPP (Spatial Pyramid Pooling) 层的出现恰好可以解决该问题, 把不同大小的 windows 经过处理后得到相同维度的 SPP 特征向量, 并把该特征向量作为全连接层的输入。

(5) SPP-Net 的最后一步与 R-CNN 的最后一步相同。假设经过候选区域映射后得到 512000 ( $2000 \times 256$ ) 个 windows, 经过 SPP 层后产生 512000 个 SPP 特征向量, 每个 SPP 特征向量经过全连接层后同样得到 512000 个输出特征向量。在训练阶段得到  $k$  个 SVM 分类器, 这里利用  $k$  个 SVM 分类器对这些输出特征向量进行分类, 并使用边框回归算法对 Bounding-box 的位置和大小进行微调。

### SPP 层原理

假设给定一张原始图像作为卷积神经网络模型的输入, 得到最后一层卷积层的特征图作为 SPP 层的输入。如图 5-4 所示, windows 对应的是原图候选区域映射到特征图中的区域, SPP 将这些不同大小的 windows 的特征映射到相同的维度, 作为全连接的输入, 就能保证每一个不同大小的 windows 具有相同的维度。

图 5-4 为 SPP 层的网络结构图, SPP 层使用了空间金字塔采样的方法。首先将大小为  $w \times h$  的 window 划分为  $4 \times 4$ 、 $2 \times 2$ 、 $1 \times 1$  的小块, 这里我们称之为 Block, 再对每个 Block 进行 Max Pooling 下采样操作。例如, 第一个 Block 有 16 ( $4 \times 4$ ) 个网格, 因此每个网格对应  $(w/4, h/4)$  的特征, 在每个网格中取最大值作为该网格的特征 (对特征进行 Max Pooling 操作), 因此第一个 Block 得到  $16 \times 256$  的特征向量 (其中 256 为特征图数量, 即特征图深度)。依次类推, 每个 window 经过 SPP 层后得到  $(21, 256)$  的特征向量, 其中  $21 = 4 \times 4 + 2 \times 2 + 1$ 。最后将所有 windows 组成的特征作为全连接层的输入。



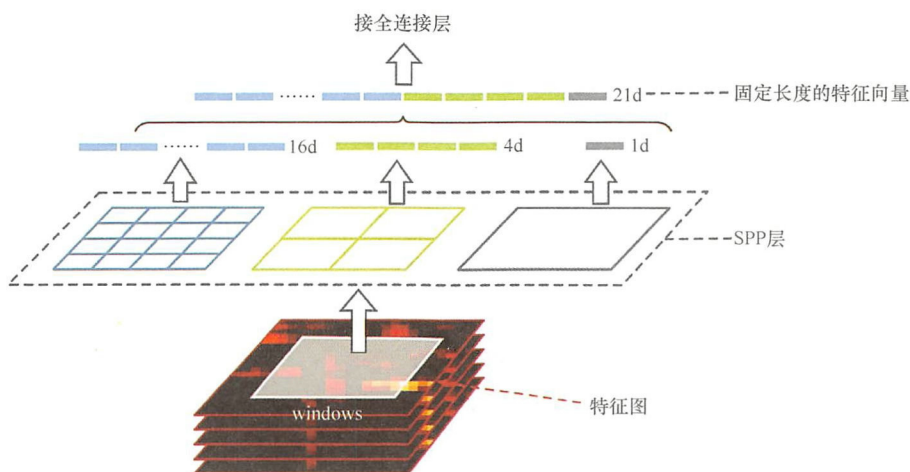


图 5-4 SPP-Layer 示例图。对特征图中进行三级金字塔 Max Pooling 操作

### SPP-Net 小结

与 R-CNN 相比较，SPP-Net 可以极大地加快目标检测的速度，但是依然存在着两个问题。

- 训练阶段步骤烦琐：与 R-CNN 一样，其训练分为微调卷积神经网络、训练 SVM 分类器、训练边框回归器 3 个单独的步骤，每次独立的训练都会产生一定的偏差；
- 训练耗时且占用空间大：与 R-CNN 一样，使用卷积神经网络和 SVM 分类器单独训练，因此需要保存大量的特征文件，占用大量的硬盘空间。

### 3. Fast R-CNN

针对 R-CNN 和 SPP-Net 的几个不足，(Ross B. Girshick, 2015) 又提出了一个更加精简的目标检测框架—Fast R-CNN。Fast R-CNN 与 R-CNN 框架模型对比，可以发现主要有两处不同：

- 最后一个卷积层后增加了 RoI Pooling 层；
- 损失函数使用了多任务损失函数 (Multi-Task Loss)，将边框回归和分类合并到卷积神经网络一起训练。

因此，Fast R-CNN 从某种意义上做到了使用深度学习的端到端训练。

图 5-5 所示为 Fast R-CNN 预测阶段的流程架构图，该网络框架流程如下。

#### Fast R-CNN 网络框架流程

- (1) 原始图像经过选择性搜索 (SS) 算法提取约 2000 个候选区域。
- (2) 将原始图像经过一次 CNN 网络，得到最后一次卷积层的特征图。



- (3) 把候选区域投影到特征图上，经过 RoI Pooling 层得到固定尺寸的特征。
- (4) RoI 特征通过两个全连接层后，分别用 Softmax 对候选区域进行分类和边框回归，微调候选框的位置与大小。

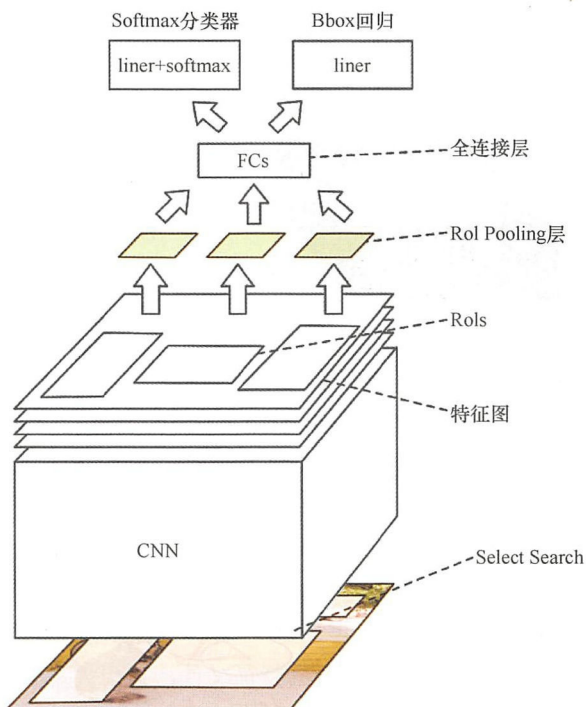


图 5-5 Fast R-CNN 网络架构图。原图经过卷积神经网络和选择性搜索 (SS) 算法得到的特征图和候选区域，每个候选区域经过 RoI Pooling 层后使用 SVM 进行预测和 Bounding-box 回归

(1) 首先，图像经过选择性搜索 (SS) 算法提取出约 2000 个候选区域框，这 2000 个候选区域基本上包括了图像中可能出现的目标物体 (Region Proposal 阶段)。

(2) 将图像输入已经训练好的 CNN 网络中，直到最后一层卷积层，在基于原始输入图像找到约 2000 个候选框位置，映射到 RoI Pooling 层。

(3) 将每一个映射后的 RoI 划分成固定大小的网格，并且对每一个小网格的所有值取其最大 (即进行 Max Pooling 操作)，得到固定大小的特征图。

(4) 将 RoI Pooling 层得到的特征图作为后续全连接层的输入，最后一层输出  $n$  个分类信息和 4 个 Bounding-box 的修正偏移量。将 Bounding-box 按照位置偏移量进行修正，再根据 NMS 算法对所有的 Bounding-box 进行筛选，即可得到对该图像的 Bounding-box 预测值以及每个 Bounding-box 对应的分类概率。

## RoI Pooling 层

实际上, RoI Pooling 层是 SPP 层的简化版。SPP 层基于空间金字塔形式, 使用了不同大小的网格块来对特征进行映射, 而 RoI Pooling 层只需要将特征图下采样到一个  $7 \times 7$  的 RoI 网格块中。例如, AlexNet 的 Conv5 后得到 256 个特征图, 并对候选区域进行映射到特征图对应的 RoI (SPP-Net 称为 windows), 该 RoI 切分成  $7 \times 7$  个网格后进行 Max Pooling 操作, 对应一个 RoI 产生 (7,7,256) 的特征向量。最终 2000 个候选框则组成 (2000,7,7,256) 的特征作为全连接层的输入。因此, RoI Pooling 层的作用非常明显, 把不同大小的 RoI 变成统一大小的特征向量, 方便实现端到端的训练。

## 多任务损失函数

R-CNN 训练过程分为 3 个阶段, 而 Fast R-CNN 直接使用 softmax 函数替代 SVM 分类, 同时利用多任务损失函数把边框回归也加入了网络中进行学习, 使得整个的训练过程实现端到端。

这里全连接层的输出有如下两个。

- **Softmax Loss**: 计算  $K+1$  类的分类损失函数 ( $K$  个目标类别, 1 个背景类别)。
- **Regression Loss**:  $K+1$  的分类相对应 Bounding-box 的 4 个坐标值。

最终将所有结果通过 NMS 算法产生最终 Bounding-box 和其所属分类。由于分类和边框回归都是基于同一个损失函数, 因此 CNN 网络训练的时候不需要把边框回归任务与 CNN 特征检测分开, 极大地加快了训练时间和减少了训练阶段的步骤。该多任务损失函数为:

$$L(p, u, t^u, v) = L_{cls}(p, u) + \lambda L_{loc}(t^u, v) \quad (5-1)$$

其中,  $L_{cls}(p, u)$  为目标分类  $u$  的对数损失,  $p = (p_1, p_2, \dots, p_K)$  为每一个 RoI 在  $K+1$  个分类中的离散概率分布:

$$L_{cls}(p, u) = -\log p_u \quad (5-2)$$

另外, 式 (5-1) 的  $\lambda \in \mathbf{R}$  为边框回归损失的权重值。  $L_{loc}(t^u, v)$  为目标分类  $u$  的边框回归损失函数:

$$L_{loc}(t^u, v) = \sum_{i \in \{x, y, w, h\}} \text{smooth}_{L1}(t_i^u, v_i) \quad (5-3)$$

式 (5-3) 中  $v = (v_x, v_y, v_w, v_h)$ , 目标分类  $u$  的边框网络预测值为  $t^u = (t_x^u, t_y^u, t_w^u, t_h^u)$ 。

$$\text{smooth}_{L1}(x) = \begin{cases} 0.5x^2 & \text{if } (|x| < 1) \\ |x| - 0.5 & \text{otherwise} \end{cases} \quad (5-4)$$

根据作者在论文中的介绍, 使用 L1 loss 比 R-CNN 和 SPP-Net 中的 L2 loss 更具

有抗噪性，对数值波动不敏感，能够更好地找到目标分类的边框值。

### Fast R-CNN 小结

Fast R-CNN 的优点为：

- 获得更高的检测精度；
- 基于多任务损失函数使得训练方式变得更加简单；
- 采用 RoI Pooling 层，方便实现卷积神经网络模型端到端的训练。

可是 Fast R-CNN 也有其缺点：

- 候选区域建议算法的提取方法使用了选择性搜索（SS）算法，该算法占用目标检测总时间的 2/3，因此还无法满足实时性要求；
- 由于候选区域建议是预先使用选择性搜索（SS）算法提取得到的，因此并没有实现真正意义上端到端的训练和预测。

## 4. Faster R-CNN

既然 Fast R-CNN 最长耗时的操作在候选区域生成的阶段，并且候选区域框的质量好坏程度直接影响后续使用卷积神经网络进行目标检测的精度。那有没有可能直接使用卷积神经网络产生的候选区域框并对其分类？针对这个问题，（Ren et al., 2015）提出了 Faster R-CNN 网络框架。

Faster R-CNN 把目标检测的 4 个基本步骤（提取候选区域、特征提取、特征分类、边框回归）统一到一个深度网络框架里面。其中候选区域的生成使用候选区域网络（Region Proposal Network, RPN）取代了 Fast R-CNN 中的选择性搜索（SS）算法，而特征提取、分类、Bounding-box 回归 3 个操作仍然沿用 Fast R-CNN 的方法。使得候选区域框的提取和 Fast R-CNN 后端融合到一个卷积神经网络模型中，首次实现了真正意义上的端到端的训练与预测。

Faster R-CNN 算法由主要两大模块组成：

- RPN 层进行候选框提取；
- 最后的分类与 Bounding-box 回归依然沿用 Fast R-CNN 的检测模块（RoI Pooling 层和多任务损失函数）。

图 5-6 所示为 Faster R-CNN 预测阶段的流程架构图，该网络框架流程如下。

### Faster R-CNN 的网络框架流程

- （1）原始图像经过卷积神经网络最后一层卷积层得到特征图，该特征图被共享用于 RPN 层和 RoI Pooling 层。
- （2）RPN 层：经过卷积神经网络得到的特征图作为 RPN 网络层的输入，用于生成候选区域框。

(3) RoI Pooling 层：输入为特征图和候选区域框，输出 RoI 特征图用作后续全连接层的输入。

(4) 最后利用候选区域框的特征图判定候选框的类别，同时使用边框回归获得 Bounding-box 的精确位置。

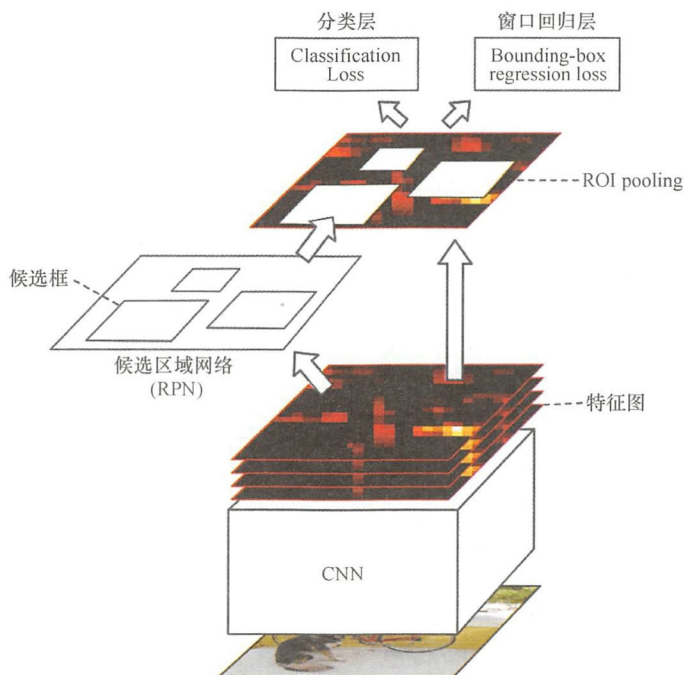


图 5-6 Faster R-CNN 架构图。输入图像经过卷积神经网络后得到特征图，特征图经过 RPN 层和 ROI 层后得到检测的 Bounding-box 目标窗口

(1) 首先，原始图像输入卷积神经网络中，得到最后一层卷积层的特征作为后续网络层的输入，该特征分为两路，被后续的 RPN 层和 RoI Pooling 层所共享（其中 RoI Pooling 层为 Fast R-CNN 的网络层）。

(2) RPN 层用于生成候选区域框，每张特征图生成多个候选区域。如果最后一层卷积层生成 256 个特征图，每张特征图上生成 300 个候选区域，那么 RPN 层共产生 76800 ( $256 \times 300$ ) 个候选区域。其目的是代替在输入图像上进行选择性搜索 (SS) 算法寻找合适的候选区域框这一耗时操作。

(3) 把 RPN 层得到的候选区域框作为 RoI Pooling 层的输入，使每个候选区域产生固定尺寸的 RoI 特征图。

(4) 最后一步与 Fast R-CNN 一样，利用 Softmax Loss 获得分类概率和 Smooth



L1 Loss 进行边框回归。假设步骤 (3) 中产生的 RoI 特征大小为  $(32, 32, 256)$ ，经过分类层输出每一个位置上 9 个候选区域 (anchor) 属于前景和背景的概率，因此分类层的输出特征为  $(32, 32, (9 \times 2))$ ；窗口回归层则输出每一个位置上 9 个候选区域对应窗口应该平移缩放参数，因此窗口回归层的输出特征为  $(32, 32, (9 \times 4))$ 。(anchor 的具体含义请继续阅读 RPN 层原理)

### 候选区域网络 (Region Proposal Networks, RPN)

Faster R-CNN 抛弃了传统的滑动窗口和选择性搜索 (SS) 算法，直接使用 RPN 网络生成候选区域，这也是 Faster RCNN 的巨大优势，能极大提升检测候选框的生成速度。

现在我们来了解候选区域 (anchor) 是如何产生的。假设最后一层卷积层得到的特征图尺寸为  $32 \times 32$ ，深度为 256，即对应特征为  $(32, 32, 256)$ 。如图 5-7 所示为一张  $32 \times 32$  的特征图，其对应应有 1024 个候选区域中心，可以产生 9 个可能的候选窗口：由 3 种面积，3 种比例  $(1:1, 1:2, 2:1)$  组成，即候选窗口通过设定的参数枚举得到。图中的红、黄、蓝色为候选区域中心产生的 9 个候选窗口示例。

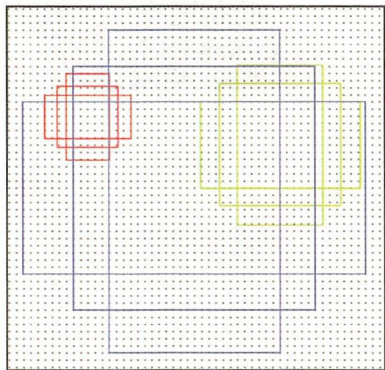


图 5-7 候选窗口 (anchor) 示例

如图 5-8 所示为 RPN 网络层，用于对目标物体和背景进行分类，并对边框进行回归。RPN 使用一个小窗口 (Sliding window) 在最后一层卷积层得到的特征图上滑动。滑动窗口为目标识别提供定位参考，区别出物体和背景的位置。对于每一个位置，Sliding window 产生  $n$  个候选区域，通过多任务损失函数回归出候选窗口的偏移量并给出每个候选窗口的分类概率。

假设输入 CNN 网络模型中的图像尺寸大小为  $512 \times 512$ ，经过最后一层卷积 Conv5 后得到 256 张  $16 \times 16$  大小的特征图。接着在这个  $16 \times 16$  大小的特征图上使用  $3 \times 3$  大小的卷积核进行卷积操作，最后同样输出 256 维的特征向量。假设 Conv5

的特征图中每个点上有  $n$  个候选窗口（默认  $n = 9$ ），而每个候选窗口分为目标物体和背景，所以每个点由 256 维特征转化为  $2 \times n$  scores；另一方面，每个候选窗口都有  $[x, y, w, h]$  对应 4 个偏移量，所以同样每个点由 256 维特征转化为  $4 \times n$  的坐标值（coordinates）。上述分别对应图 5-8 中的两个分支全连接层。

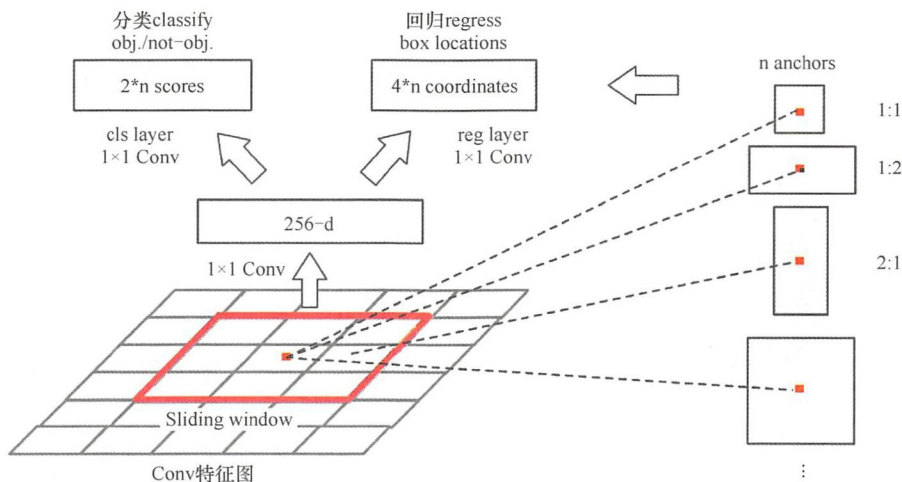


图 5-8 RPN 网络

### Faster R-CNN 小结

Faster R-CNN 将分离的候选区域建议算法与卷积神经网络的分类回归网络融合在一起，使用端到端的卷积神经网络模型进行目标检测，在速度和精度上都有所提升。然而 Faster R-CNN 还达不到实时目标检测的要求。

总的来说，从 R-CNN、SPP-Net、Fast R-CNN，到 Faster R-CNN，基于深度学习目标检测的流程变得越来越精简，精度越来越高，速度也越来越快。可以说基于区域选择的网络模型是当前目标检测算法中重要的分支。下面来介绍目标检测算法的另一个分支——基于回归的网络模型。

## 5.1.3 基于回归的网络

### 1. YOLO 网络模型

上面讲述的 R-CNN、SPP-Net 和 Faster R-CNN 模型架构都是基于区域选择的目标检测方法，其发展是候选区域算法从一开始的原图搜索到使用特征图中的候选区域作为代替。Faster R-CNN 有较高的检测精度，但是其速度还远远不能满足实时性的要求。而基于回归方法的深度学习目标检测算法则开始凸显其重要性，该类方法

使用了回归的思想，输入图像后直接在其不同位置上回归出这个位置的目标边框以及目标类别。

YOLO 没有选择基于候选区域算法的方式对网络进行训练，而是直接采用全图进行训练的模式。其好处在于可以更方便快捷地区分目标物体和背景区域，缺点是在提升检测速度的同时牺牲了精度。

YOLO 网络结构

图 5-9 为 YOLO 预测阶段的流程架构图，该网络框架流程如下。

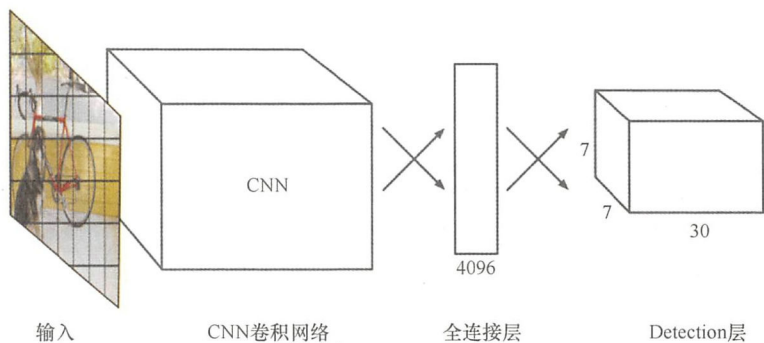


图 5-9 YOLO 网络结构

YOLO的网络框架流程

- (1) 将输入图像缩放成固定大小（如  $448 \times 448$ ），并对其分割成  $S \times S$  个网格。
- (2) 原始图像经过 CNN 网络，直到卷积神经网络的第一层全连接层（FC1）。
- (3) 原图中每个网格负责预测  $B$  个 Bounding-box，每个 Bounding-box 有 4 个位置信息和 1 个物体分类的概率。另外每个网格还负责预测  $C$  个分类，最终得到  $S \times S \times (B \times 5 + C)$  大小的特征矩阵，并对该特征矩阵进行 Softmax 分类（Detection 层）。
- (4) 根据阈值去除物体分类概率较低的窗口，最后使用 NMS 算法去除冗余窗口，得到目标窗口。

- (1) 首先把原图缩放成固定大小作为卷积神经网络模型的输入。假设 YOLO 网络的输入图像大小为  $448 \times 448$ ，对原图分割成  $7 \times 7$  个网格，并记录下这些数据。
- (2) 原始图像经过卷积神经网络模型后，继续把特征传递给全连接层（FC1、FC2 层），这里与传统的卷积神经网络架构一样，在 FC2 层输出 4096 个神经元。
- (3) Detection 层：在步骤（1）中将原图分割成  $7 \times 7$  个网格，每个网格负责预测  $B$  个 Bounding-box，每个 Bounding-box 有 4 个位置信息和 1 个物体分类概率。假设  $B$  为 2，则一共产生  $10 (2 \times 5)$  个参数。另外每个网格还负责预测  $C$  个分类，假

设有 21 个分类，则一个网格对应的向量为 31 ( $2 \times 5 + 21$ )。因此，最终得到  $7 \times 7 \times 31$  大小的特征矩阵。实际上，我们可以将 FC2 层的 4096 个神经元理解为 Detection 层的输入，Detection 层对输入数据进行卷积，产生  $(7, 7, 31)$  大小的特征图，其中张量  $(7, 7, 31)$  对应  $S \times S \times (B \times 5 + C)$ 。

(4) 最后根据张量  $(7, 7, 31)$  的特征中代表物体分类概率的较低值进行过滤，然后使用 NMS 算法去除冗余窗口，得到最终的目标窗口。

### YOLO 模型思想

图 5-10 所示为 YOLO 网络架构模型思想，首先需要将输入的图像（例如  $448 \times 448$ ）分割为  $S \times S$ （例如  $S = 7$ ）个网格，因此得到每一个网格大小为  $64 \times 64$ 。如果一个物体的中心落在某网格内，则相应网格负责检测该物体。每个网格要预测  $B$ （例如  $B = 2$ ）个 Bounding-box，每个 Bounding-box 对应 5 个预测参数，其中包括 4 个位置信息  $[x, y, w, h]$ ，另外附带预测一个置信度（confidence）值：

$$confidence = P_r(Object) * IOU(truth|pred) \quad (5-5)$$

其中， $confidence$  代表所预测的 Bounding-box 中含有目标的置信度  $P_r(Object)$  和该 Bounding-box 预测准确率  $IOU(truth|pred)$ 。如果有目标物体落在一个网格里， $P_r(Object)$  为 1，否则为 0。第二项  $IOU(truth|pred)$  用于预测 Bounding-box 和实际的 ground truth 之间的 IOU 值，如果存在物体则计算出 IOU。同时会预测存在物体的情况下该物体属于某一类的后验概率  $P_r(Class_i|Object)$ 。

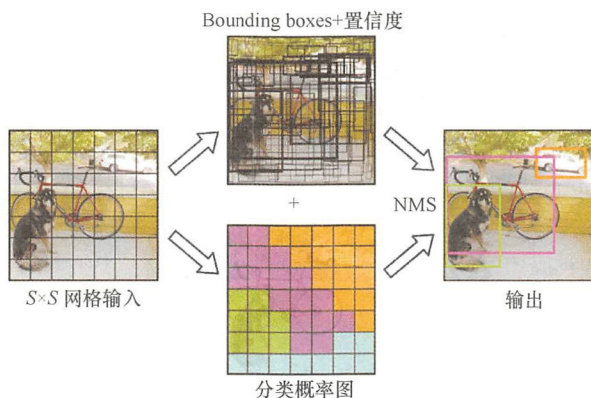


图 5-10 YOLO 网络架构模型思想

另外，每个网格还要预测一个类别信息  $C$ 。 $S \times S$  个网格中的每个网格要预测  $B$  个 Bounding-box 和  $C$  个类别。因此输出就是  $S \times S \times (B \times 5 + C)$  的特征。在预测分类时，每个网格预测的分类信息和 Bounding-box 预测的置信度相乘，得到每个 Bounding-box 的得分：



$$\begin{aligned}
&P_i(Class_i|Object)*P_i(Object)*IOU(truth|pred) \\
&=P_i(Class_i)*IOU(truth|pred)
\end{aligned} \tag{5-6}$$

等式 (5-6) 左边第一项为每个网格预测的类别信息, 其中  $P_i(Class_i)$  为第  $i$  个分类的概率。这个乘积既包括了预测的 Bounding-box 属于某一类的概率, 也包括了该 Bounding-box 准确度的信息。得到每个 Bounding-box 的分类得分后, 设置阈值滤掉得分较低的 Bounding-box, 对余下的 Bounding-box 进行 NMS 处理, 得到最终的检测结果。

### YOLO 损失函数

对于任何一种网络, 损失函数的定义决定着网络训练的效果, 而 YOLO 是多任务损失函数, 其主要考虑了以下因素。

#### (1) Bounding-box 的坐标预测误差

值得注意的是, 在 YOLO 框架中 Bounding-box 的坐标  $(x, y, w, h)$  并不是指每个 Bounding-box 的起始坐标和窗口宽度,  $x, y$  是相对于该网格的偏移,  $w, h$  是该网格对应于整张图像的长宽比例。对不同大小的 Bounding-box, 小的 Bounding-box 偏移量级与大的 Bounding-box 偏移量是不相同的 (小的 Bounding-box 预测偏移一点对结果影响很大, 相对而言, 较大的 Bounding-box 大小预测偏移一点对结果影响可能不是太大)。为了解决这个问题, 作者在损失函数中对  $w$  和  $h$  求平方根进行回归:

$$\begin{aligned}
\text{loss}_{\text{coord}} = & \sum_{i=0}^{S \times S} \sum_{j=0}^B \Pi_{ij}^{\text{obj}} (x - \hat{x}_i)^2 + (y - \hat{y}_i)^2 \\
& + \sum_{i=0}^{S \times S} \sum_{j=0}^B \Pi_{ij}^{\text{obj}} (\sqrt{w} - \sqrt{\hat{w}_i})^2 + (\sqrt{h} - \sqrt{\hat{h}_i})^2
\end{aligned} \tag{5-7}$$

#### (2) Bounding-box 的置信度预测误差

对于 Bounding-box 的损失函数设计, 需要计算包含目标和非目标的损失:

$$\text{loss}_{\text{conf}} = \sum_{i=0}^{S \times S} \sum_{j=0}^B \Pi_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 + \sum_{i=0}^{S \times S} \sum_{j=0}^B \Pi_{ij}^{\text{no-obj}} (C_i - \hat{C}_i)^2 \tag{5-8}$$

#### (3) 分类预测误差

分类预测误差主要用于判断 Bounding-box 的所属类别, 其损失函数为:

$$\text{loss}_{\text{classes}} = \sum_{j=0}^B \Pi_{ij}^{\text{obj}} \sum_{c \in C} (p_i(c) - \hat{p}_i(c))^2 \tag{5-9}$$

值得注意的是, 一个网格只预测一次类别, 即默认每个网格中的  $B$  个 Bounding-box 都属于同一类别。这也是 YOLO 框架的不足之处, 最多只能预测  $S \times S$  个分类。

假设  $\lambda \in \mathbf{R}$  为各个误差值的权重, 最终 YOLO 的多任务损失函数为:

$$\text{loss} = \lambda_{\text{coord}} \times \text{loss}_{\text{coord}} + \lambda_{\text{conf}} \times \text{loss}_{\text{conf}} + \lambda_{\text{classes}} \times \text{loss}_{\text{classes}} \quad (5-10)$$

### YOLO 小结

YOLO 将目标检测任务转换成一个回归问题，大大加快了检测的速度，使得 YOLO 检测最快速度达到 200fps。由于每个网络预测目标窗口时使用了全图信息，其检测召回率上升。其缺点是：由于免去了候选区域建议阶段，只使用  $S \times S$  个网格进行回归，制约目标检测精度和限制小物体的检测（如一个网格中出现了两个小物体的中心，最终 YOLO 最多只能检测出其中一个物体）。

#### Intersection Over Union, IOU

模型产生的目标窗口和原来标记窗口的交叠率，即检测结果（Detection Result）与 Ground Truth 的交集比上它们的并集，即为检测的准确率 IOU：

$$\text{IOU} = \frac{\text{Detection\_Result} \cap \text{Ground\_Truth}}{\text{Detection\_Result} \cup \text{Ground\_Truth}} \quad (5-11)$$

## 2. SSD 网络模型

上一节中我们分析了 YOLO 框架模型存在的问题，使用全图的特征在  $S \times S$  的网格内对物体进行回归定位的目标检测方法在一定程度上限制了目标检测的准确率。而基于候选区域的方法能保证较高的定位准确率，那么能否结合 YOLO 的回归思想和 Faster R-CNN 的候选区域网络（RPN）的方法来进行目标检测呢？

2016 年（Wei Liu et al.）提出了 SSD 目标检测框架，其结合了 YOLO 的回归思想以及 Faster R-CNN 的候选区域（anchor）机制，使得基于深度学习的目标检测算法做到了实时检测，并进一步提高了其检测准确率。作为目前主流的目标检测框架之一，SSD 相比 Faster R-CNN 有明显的速度优势，相比 YOLO 又有明显的 mAP 优势，其主要特点有：

（1）从 YOLO 中学习了把目标检测方法转化为回归的思路，实现端到端的训练与预测；

（2）基于 Faster R-CNN 的候选区域机制之上，提出了相似的 Prior box 概念；

（3）利用基于特征金字塔（Pyramidal Feature Hierarchy）的检测方式。

### SSD 网络结构

（1）图 5-11 所示为 SSD300 的网络结构图。把原始输入图像缩放为固定大小，如  $300 \times 300$  作为卷积神经网络模型的输入，其基础网络模型使用了 VGG-16。

（2）SSD 把 VGG16 的 FC6 层、FC7 层的全连接方式改为卷积层，然后继续向后添加若干卷积层直到 Conv10\_2 层，最后接一个 Pooling 层（Pool 11）。

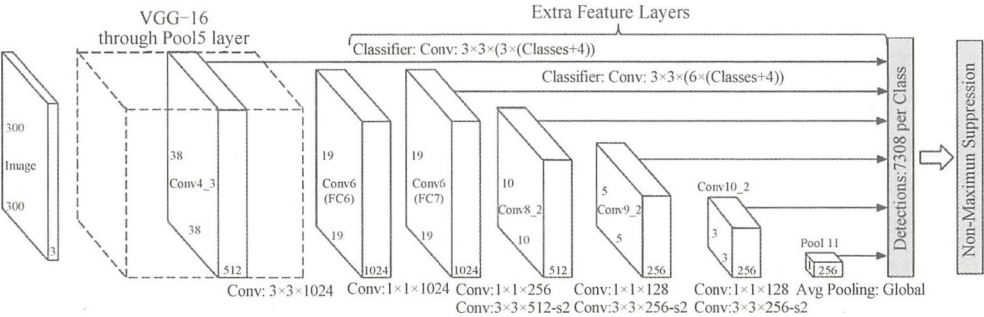


图 5-11 SSD 网络结构图

- (3) 定义完网络结构后，SSD 采用了特征金字塔结构对物体进行回归检测，即检测时会利用多个不同卷积层产生的特征图。如利用修改过后 VGG16 的 Conv4\_3，Conv\_7，Conv6\_2，Conv7\_2，Conv8\_2，Conv9\_2 这些不同卷积层得到的特征图。
- (4) 提取特征层作为 Detection 层的输入。在不同尺寸的特征图上产生多个 Prior box，假设特征图大小为  $m \times m$ ，一张特征图上产生了  $k$  个 Prior box，一个 Pror box 对应  $c$  个分类和 4 个边框信息，因此产生维度为  $m \times m \times k \times (c+4)$  的特征，最后在这些特征图产生的特征上进行分类和边框回归。
- (5) 利用 NMS 算法对最后结果进行筛选。
- 其网络框架流程如下。

SSD的网络框架流程

(1) 把图像缩放成固定大小（例如 300×300）作为卷积神经网络的输入。

(2) 在卷积神经网络最后的卷积层继续添加多次卷积层，并保存卷积后得到的特征图。

(3) 经过不同卷积层后得到的特征图上取得多个 Prior box，用于进行物体分类和边框回归（Detection 层）。

(4) 使用 NMS 非极大值抑制算法选出最终候选框。

**SSD 模型核心 Prior box**

下面来看如何根据提取到的特征图，建立检测目标位置和其特征的对应关系。这里 SSD 基于 Faster R-CNN 的候选区域机制，提出了 Prior box 的概念。

如图 5-12（b）所示，假设某卷积层后的特征图的大小为  $8 \times 8$ ，使用  $3 \times 3$  的卷积核提取特征图网格中每一个中心位置的特征，然后在这个特征向量上进行回归，得到目标的窗口信息和类别信息。由于 SSD 采用了多层次特征图，可以感受到不同尺寸的特征图上的特征，因此可以利用浅层网络的特征图检测小物体，利用深层网



络的特征图检测大物体。

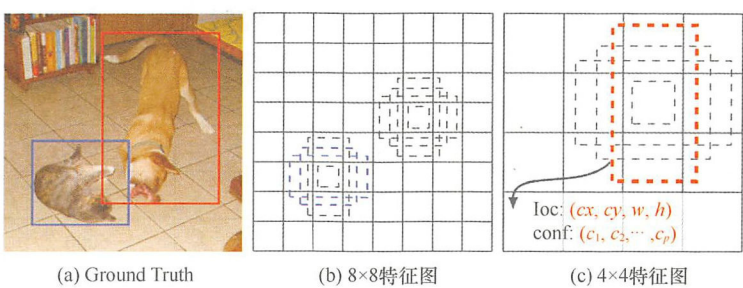


图 5-12 SSD 的 Prior Box。(a) 为 Ground Truth，(b) 为 8×8 大小的特征图，(c) 为 4×4 大小的特征图。对比 (b)、(c) 图可以看出特征图尺寸越大，产生的 Prior box 可以检测到小物体，而特征图尺寸越小，产生的 Prior box 映射回原图中只能检测较大的物体

SSD 与 YOLO 类似，每一个 Bounding-box 对应的特征图网格中心位置是固定的，与 Faster R-CNN 类似，Prior box 是通过枚举产生的。对于每一个网格中心位置上，我们需要预测最终得到的 Bounding-box 与 Prior box 之间的偏移量  $[x,y,w,h]$ ，以及每一个 Bounding-box 的分类得分。假设一个中心位置有  $k$  个 Prior box，那么需要计算出  $c$  个类别得分，和该 Bounding-box 相对于 Prior box 的 4 个偏移量。最后对于特征图网格中心位置上产生  $k \times (c+4)$  个特征向量，对于一张  $m \times m$  大小的特征图，则产生  $m \times m \times k \times (c+4)$  的特征。

### SSD 小结

SSD 目标检测方法的优点明显，运行速度可以与 YOLO 媲美，检测精度则可以与 Faster RCNN 媲美。但其缺点是需要人工设定 Prior box 的参数值：网络中 Prior box 的基础大小和形状不能直接通过学习获得，需要手工设置。而网络中不同特征图使用的 Prior box 大小和形状都不一样，导致调试过程依赖于实际经验。

## 5.1.4 目标检测小结

基于深度学习的目标检测方法主要分为基于候选区域建议系列的目标检测框架，和基于回归的目标检测框架。自从深度学习被引入图像处理领域之后，目标检测在计算机视觉领域获得了巨大的进步。

表 5-1 所示将利用深度学习的目标检测算法进行了对比（基于 VOC2007 数据集）。从表中我们得知，速度最快的是使用了回归方法的 tiny YOLO，但其以损失准确率来换取超实时的时间，只使用了 98 个 Bounding-box，因此对小物体的检测效果一般。而效果最好的是 SSD512，输入  $512 \times 512$  大小的图像，并对 24564 个 Bounding-box



进行回归计算，最后得到 76.8 的 mAP 和速度上达到了准实时速度 19fps。

表 5-1 基于深度的目标检测算法对比

方法	mAP	FPS	batch size	Boxes	输入
R-CNN	66.0	47s	1	~ 2000	~
SPP-Net	66.9	0.80s	1	~ 2000	~
Fast R-CNN	66.9	0.32s	1	~ 2000	~
Faster R-CNN	73.2	7	1	~ 6000	~
tiny YOLO	52.7	155	1	98	448×448
YOLO	66.4	21	1	98	448×448
SSD300	74.3	46	1	8732	300×300
SSD512	76.8	19	1	24564	512×512

除此之外，学者们基于上述框架从其他方面提出了一些提高目标检测性能的方法。

1. 难分样本挖掘（Hard Negative Mining）

难分样本，即非目标框的背景数据，一般产生用于训练的正负样本比例是非常不平衡的（负样本占据了大量比例）。当负样本居多时，直接训练会导致网络过于重视负样本，从而导致损失不稳定。为了更好地利用有效数据，（Shrivastava, 2016）将难分样本挖掘机制嵌入 SGD 算法中，使得 Fast R-CNN 在训练的过程中根据候选区域的损失自动选取合适的候选区域框作为正负例训练。实验结果表明，使用该机制可以使得 Fast R-CNN 算法在 VOC2007 和 VOC2012 上 mAP 提高 4% 左右；另外，SSD 在训练时会依据置信度得分对 Prior box 进行排序，挑选置信度较高的 Prior box 进行训练，并控制正负样本比例为 1：3。

2. 多层特征融合

Fast R-CNN 和 Faster R-CNN 都只是利用最后的卷积层产生的特征进行目标检测，但由于深层网络的卷积层特征经过高度抽象，其特征损失了较多细节。（T Kong et al, 2016）使用 Hypernet 等方法充分利用卷积神经网络的多层特征融合进行目标检测，除了利用高维的卷积层特征之外，同时结合低维的卷积层特征进行目标检测，以便更好地利用低维特征的纹理信息，进一步提升目标检测效果。

3. 结合上下文特征进行目标检测

根据（Gidaris et al. 2015）和（Bell et al. 2016）得知，某一类物体周围出现的内容信息将能够有效地对其位置进行锁定（如椅子旁边很可能有桌子）。因此在提取候选区域框的特征用于目标检测时，结合候选区域的上下文信息，可能会给我们带来更多意想不到的效果。

## 5.2 图像语义分割

在自动驾驶系统中的街景识别与场景理解、无人飞行器对航片进行分析和物体避障，以及其他消费级 AI 终端中，图像语义分割（Semantic image segmentation）都起着举足轻重的作用。图像语义分割作为计算机视觉中图像理解（Image understanding）的重要一环，不仅在工业界的需求日益凸显，同时也是学术界的研究热点之一。

实际上，图像是由众多像素点组成的矩阵，而图像语义分割就是将像素按照图像中表达不同的语义进行分组（Grouping）/ 分割（Segmentation），即对图像上的每一个像素点进行分类。如图 5-13（b）所示的多边形框为对图 5-13（a）中的目标物体进行语义分割。



图 5-13 图像语义分割。(a) 为原始输入图，(b) 为经过图像语义分割算法获得的分割图。

图像语义分割能够对物体进行像素级别的分类，精细切割出物体所在的像素位置

在 2015 年之前，如何有效地把图像进行高效快速的语义分割仍然是计算机图像的研究热点。（Jonathan Long, 2015）使用深度学习技术成功对图像进行语义分割后，为这一领域开启了新的篇章。其图像分割平均精度（mIoU）从 2015 年 FCN（Jonathan Long, 2015）的 62.2%，到 DeepLab（Chen L C, 2016）框架的 72.7%，后来到牛津大学的 CRF as RNN（Zheng S, 2016）的 74.7%。目前图像语义分割的精度还没能超过 95%，因此该领域方向仍旧有很大的进步空间，或许正在书前的你将会成为下一个图像语义分割的领军人物。下面让我们一起去体会图像语义分割的精彩吧！

### 5.2.1 传统图像分割方法

传统的图像分割方法主要分为以下几类：

- 基于阈值的分割方法；

- 基于区域的分割方法;
- 基于聚类分割方法;
- 基于图论的分割方法。

### 1. 基于阈值分割

利用阈值对图像进行分割是最直观方便的一种方法,优点是计算简单、运算效率较高。此方法在重视运算效率的应用场合中得到了广泛应用。按使用阈值情况,可以分为全局阈值分割和局部阈值分割。

阈值分割方法主要由输入图像  $f$  到输出图像  $g$  作如下变换:

$$g(i, j) = \begin{cases} 1 & f(i, j) \geq T \\ 0 & f(i, j) < T \end{cases} \quad (5-12)$$

其中,  $T \in \mathbf{R}$  为阈值,  $f(i, j)$  为图像对应的像素值。由此可见, 阈值分割算法的关键是确定阈值, 将阈值与像素点的灰度值进行比较, 分割出图像的前景色和背景色。

全局阈值通过设置一个统一的阈值对全图进行分割。但是图像会受光照不均匀的影响, 使得待分割的目标区域灰度变换很大, 这时全局阈值不再适用, 需要通过设置合理的图像窗口, 对每个窗口使用局部阈值进行分割, 最后再将所有局部分割结果进行融合。

### 2. 基于区域分割

基于区域的分割方法是利用图像分割目标的区域性质, 基于此产生了很多优秀的分割算法。这里主要介绍区域分割算法的两个不同特性: 区域生长合并的方法和区域分裂合并的方法。

(1) 区域生长合并: 将具有相似性质的像素集合起来构成区域。具体先在每个需要分割的区域找一个种子像素作为生长的起点, 然后将种子像素周围邻域与种子像素有相似性质的像素合并到种子像素所在的区域中。不断重复上述过程, 直到没有满足条件的新像素为止。其优点是计算简单, 对于较均匀的连通目标有较好的分割效果; 缺点是需要人为确定种子像素和相似性规则, 并对噪声敏感, 另外当分割目标较大时, 其分割速度较慢。

(2) 区域分裂合并: 从全图出发不断分裂得到多个子区域, 然后把各个分割子区域合并实现目标提取。该方法的关键是分裂合并准则的设计, 优点是对复杂图像的分割效果较好, 缺点是分裂合并准则设计麻烦, 算法时间复杂度大, 分裂过程中还可能破坏区域的边界。

### 3. 基于聚类分割

基于聚类的分割方法是针对每个像素和相邻像素之间的关系, 以图形中的某像素作为聚类点, 按照指定的簇数对相邻像素进行归类, 重新获得中心聚类点。不断



重复上述过程，把图像中所有像素分割为不同的聚类簇。

其中 K-means 算法广泛用于机器学习领域中的聚类分析，我们可以使用 K-means 算法对图像进行分割。算法首先选定  $K$  个聚类中心点，将图像中的某一像素坐标  $(x,y)$  以及像素值  $(R,G,B)$  共 5 个参数归一化为一个特征向量，计算该像素的特征与  $K$  个初始聚类像素点的特征之间的欧式距离，接着将其归类到与其距离最小的初始聚类中。迭代上述步骤，直到新旧类均值之差小于一定阈值。

#### 4. 基于图论分割

传统的基于图论的语义分割方法用于图像处理，主要是将图像抽象为图 (Graph) 的形式，利用图的分割方法进行图像分割。图的基本形式为：

$$G = (V, E) \quad (5-13)$$

其中  $G$  为图 (图像)， $V$  为图的节点 (像素点)， $E$  为图的边 (像素相邻关系)，边的权值为像素的相似度。利用图的分割方法对图像进行分割较为经典的方法有 GraphCut 和 GrabCut，其基本思想都是采用图论中的最小割 (Minimum Cut) 来实现图的分割，并用最大流 (Max-flow) 来获取这个最小割。

## 5.2.2 全卷积神经网络

在没有使用深度学习做图像语义分割之前，大部分研究者的工作是根据图像像素自身的低阶视觉信息来进行图像分割。这样的方法虽然计算复杂度不高，但却不能有效地分割图像中较为复杂的场景。另外，即使正确地分割出目标物体后，仍然不能标注出分割物体所属的分类，还需要经过一系列算法对分割目标进行语义提取。因此传统图像的语义分割效果不能令人满意。

在计算机视觉迎来深度学习之后，图像语义分割也进入了全新的发展阶段，以全卷积神经网络 (Fully convolutional networks, FCN) 为代表的基于卷积神经网络的图像语义分割方法相继被提出，并再一次刷新图像语义分割的精确率。

早期使用深度学习对图像进行语义分割方法是：以图像中包含某像素的一个图像块作为卷积神经网络的输入，得到该图像块的分类作为该图像块中心像素的所属类别。该方法的优点是很好地使用了深度学习中图像的高维特征，缺点如下。

(1) 存储和运行开销大。例如图像大小为  $400 \times 400$ ，对每个像素使用大小为  $15 \times 15$  的图像块在原图像上滑动窗口，使用 Same Padding 的滑窗方式将产生 160000 ( $400 \times 400$ ) 个  $15 \times 15$  大小的窗口给卷积神经网络进行分类，其存储空间和运算时间会急剧上升。

(2) 计算效率低下。因为相邻图像块的纹理特征在很大程度上是重复的。

(3) 图像块大小限制了感知区域。由于图像块大小比一般原始图像小得多 (例如  $15 \times 15$ )，因此只能提取局部特征，从而导致分类的准确性受到限制。



全卷积神经网络则是从输入图像的高维特征中恢复出每个像素所属的分类，将卷积神经网络从图像级别的分类进一步延伸到像素级别的分类（如图 5-14 所示）。

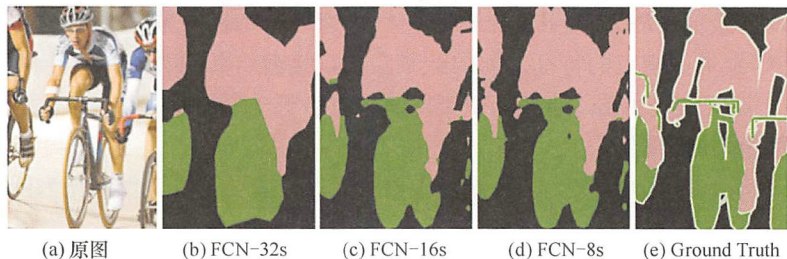


图 5-14 FCN 对图像进行语义分割示例图。Ground Truth 中包含了“人”和“自行车”两个分类，从 FCN-32s 出现了基本的语义分割效果，到 FCN-16s 和 FCN-8s 其语义分割精度进一步提高

一般而言，卷积神经网络在最后一层卷积层后会接上若干个全连接层（例如 FC1、FC2），目的是将卷积层产生的特征图通过全连接的方式，映射成固定长度的特征向量，然后通过输出层输出该图像的高维特征向量所对应的分类概率。例如 AlexNet 网络模型使用了两个长度为 4096 的全连接层（FC6、FC7），最后接一个长度为 1000 的 Softmax 输出层用于计算 1000 个类别的概率。

FCN 全称为全卷积网络（Fully Convolutional Networks），顾名思义是把普通卷积神经网络最后的全连接层换成卷积层，使得整个卷积神经网络结构中只有卷积层和 Pooling 层（没有全连接层），因此称为全卷积网络。

### FCN 网络结构

FCN 将普通卷积神经网络模型中的全连接层转换成卷积层。图 5-15 所示为 FCN 网络架构图，该神经网络的原始模型采用 AlexNet 网络结构，输入大小为  $224 \times 224$  的图像进行训练。根据第 4 章示例 2 中 AlexNet 的模型，AlexNet 经过了 5 次 Pooling 层，最后产生的特征图大小为  $(7, 7, 256)$ ，原本 Pooling 层后接的 FC1 全连接层，也就是把特征图的  $(7, 7, 256)$  维特征与 4096 个神经元进行全连接。FCN 把 FC1 层转化为卷积层，卷积核大小为  $1 \times 1$ ，因此通过 FC1 卷积层后特征由  $(7, 7, 256)$  变为  $(1, 1, 4096)$ 。可见 FCN 把全连接层转化为卷积层后，该层的参数仍然为 4096，改变了网络参数的结构方式而没有改变网络中的权重参数。同理，FC2 全连接层原输入为 4096 个神经元，输出也为 4096 个神经元，FCN 继续把 FC2 层转换为卷积层，卷积核大小为  $1 \times 1$ ，因此对应 FC2 卷积层的权重参数为  $(1, 1, 4096)$ 。最后是 Softmax 输出层，原 Softmax 输出层的输入为 4096 个神经元，输出为 1000 个神经元（1000 个分类），FCN 把 Softmax 层转换为卷积层，同理卷积核大小为  $1 \times 1$ ，因此对应 Softmax 层的权重参数为  $(1, 1, 1000)$ 。

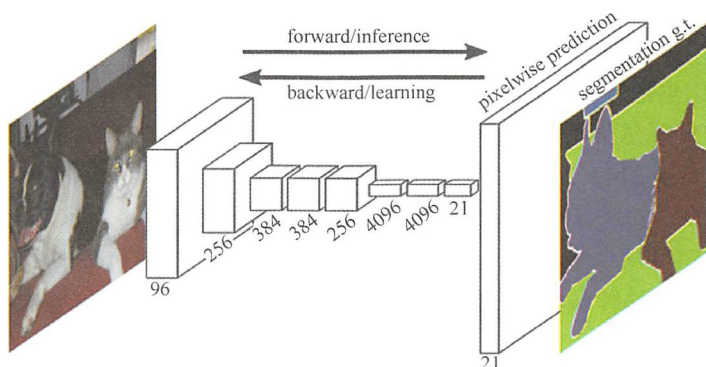


图 5-15 FCN 架构图。该 FCN 网络架构以 AlexNet 为例，一共有 8 个卷积层、5 个 Pooling 层，使用 Pascal 数据集有 21 个类别，因此最后的卷积层深度设定为 21

继续以 FCN-32s 为例，假设图 5-15 中 FCN 网络的输入图像为  $512 \times 512$ ，经过 FCN 最后一层卷积层生成的热图（Heat Map）的特征大小为  $(16, 16, 21)$ ，其中 16 为 Heat Map 的长和宽、21 为热图对应的深度（即 21 个类别）。把热图经过 32 倍数上采样后变为  $(640, 640, 21)$  的特征，因此对应图像中的每一个像素都有 21 个值（21 个分类），21 个值中概率最高的作为该像素所属类别。如图 5-16 所示，FCN 把原来 CNN 用于预测 21 个分类的值，转换为预测热图中一个像素上 21 个分类的得分。

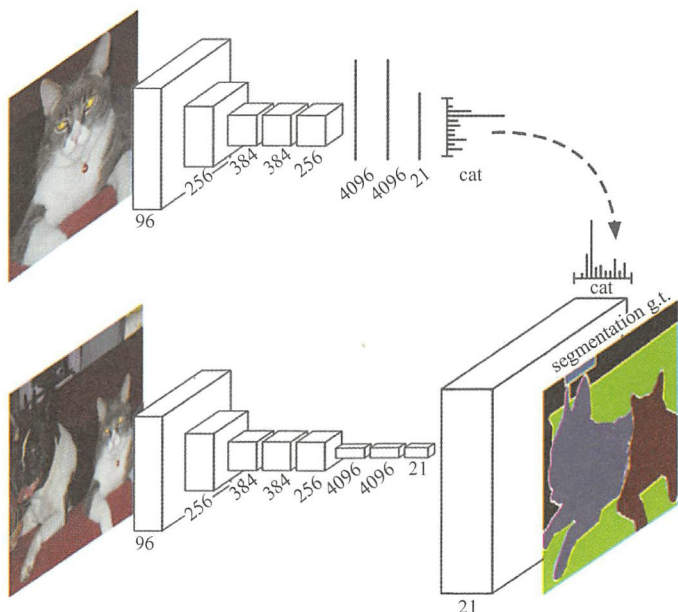


图 5-16 FCN 进行像素级别的分类预测

其中 FCN-32s 的网络框架流程如下。

### FCN-32s 的网络框架流程

- (1) FCN 训练阶段与一般的卷积神经网络训练阶段一致。把分类图像缩放成固定大小（例如  $214 \times 214$ ）输入卷积神经网络中进行学习训练，并保存训练结果。
- (2) 修改卷积神经网络最后的全连接层为卷积层，并称其为 FCN 网络。
- (3) 预测阶段会输入图像通过 FCN 网络，数据在 FCN 网络中向前传播到最后的卷积层产生的特征图称为热图，对最后产生的热图进行 32 倍上采样操作恢复成原图大小。
- (4) 最后计算上采样之后的图逐像素，求得每个像素所属类别得分。

### 上采样层和跳跃层

**上采样 (Upsampling):** 众所周知，Pooling 操作是对输入的矩阵进行降采样，而缩小输入矩阵尺寸的同时进一步提取矩阵中的高维特征。通过 FCN 网络产生的热图为一尺寸很小的图像，为了得到和原图相同大小的图像和补充图像细节，我们需要对原图进行上采样操作。FCN 原文中的上采样操作可以使用双线性插值对热图进行放大操作，也可以使用训练过的卷积神经网络进行反卷积操作。

**跳跃层 (Skip Layers):** 跳跃层用于优化结果，兼顾图像的局部和全局信息。如果将全卷积后产生的热图直接上采样 32 倍后，那么得到的分割图像较为粗糙 (FCN-32s)。因此 Jonathan 在论文中将不同 Pooling 层产生的热图进行局部上采样后融合，得到更准确的分割图。

图 5-17 所示为 FCN 的跳跃层示例图。

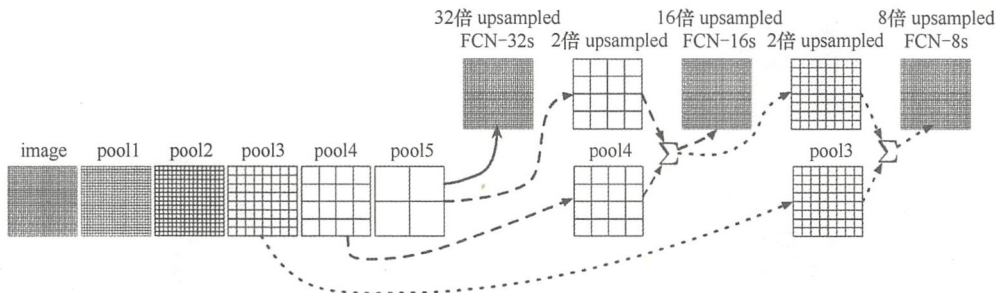


图 5-17 FCN 的跳跃层示例图

(1) FCN-32s: 直接对 Pool5 产生的热图进行 32 倍上采样操作，恢复为输入图像的大小。

(2) FCN-16s: 先对 Pool5 产生的热图进行 2 倍上采样操作，得到的结果与



Pool4 的热图的尺寸相同，然后对 Pool5 上采样的热图和 Pool4 产生的 Heap Map 两层进行矩阵求和，再进行 16 倍的上采样操作，恢复为输入图像的大小。

(3) FCN-8s：同理，FCN-8s 通过对 Pool5 上采样的热图和 Pool4 产生的 Heap Map 两层进行矩阵求和的热图进行 2 倍上采样操作，然后与 Pool3 产生的热图再次进行矩阵求和，最后进行 8 倍上采样操作，恢复为输入图像的大小。

### FCN 小结

FCN 开创性地使用深度学习对图像进行端到端的图像语义分割操作，极大地提高了图像语义分割的时间和精度。但正是因为是第一次尝试，因此会存在如下一些不足。

- FCN 的 mAP 和 IOU 的准确率仍然有待提升：如图 5-14 所示，FCN-8s 的效果明显好于 FCN-32s 的效果，但是与 Group Truth 对比还是比较粗糙，对图像细节还原度不足。
- 没有充分考虑相邻像素间的关系：由于 FCN 是对各个像素进行分类，因此没有充分考虑相邻像素间的关系，缺乏空间一致性。

## 5.2.3 SegNet 网络

牛津大学 (Badrinarayanan, 2015) 提出了一个基于编码和解码的网络模型 SegNet，该网络模型的对称设计较为优雅和易于理解。如图 5-18 所示，使用 SegNet 对中国的某高速公路进行测试，SegNet 能够正确地识别出一台后装奇特的泥头车，连道路上细小的道路线也能够正确识别出来，另外道路上的可行驶区域 mIOU 超过了 75%。

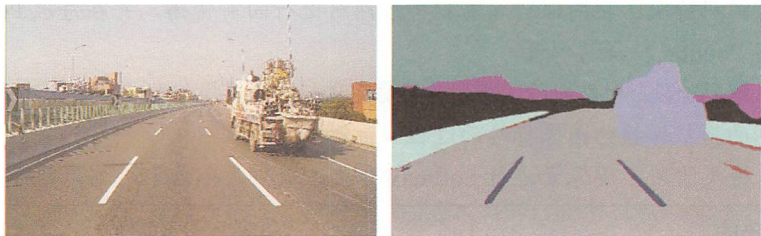


图 5-18 直接使用 SegNet 对公路场景进行语义分割

### SegNet 网络模型

SetNet 网络模型借鉴了自编码的思想，先对图像进行编码，再进行解码。核心由编码 (Encoder) 阶段和解码 (Decoder) 阶段组成。

- 编码阶段：将图像转化为特征，产生低分辨率的图像作为特征映射的过程。



- 解码阶段：将特征转化为图像标签，低分辨率图像映射到像素级标签的过程。

图 5-19 所示的左半部分为 SegNet 的编码阶段，在编码阶段使用 VGG16 网络通过卷积层和 Pooling 层获得输入图像的高维特征。与之对应为图 5-19 的右半部的解码阶段，该阶段的卷积层和上采样（Upsample）层与编码阶段一一对应，上采样层恢复部分图像特征，其中卷积层的卷积核则是通过网络训练得到的。经过解码阶段得到与输入原图大小一致的特征图后，使用 Softmax 层检查每一像素位置对应特征深度的最大值作为所属类别。

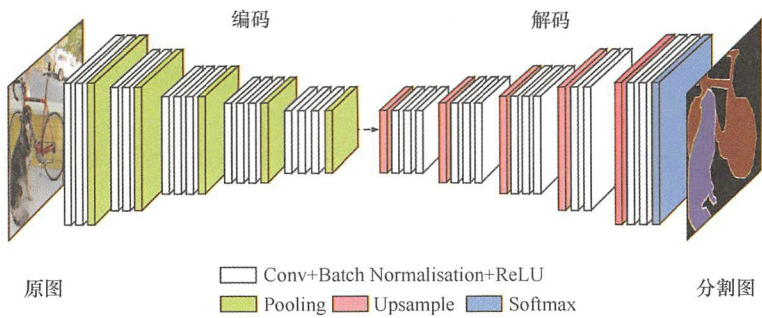


图 5-19 SegNet 架构图。该网络架构使用了卷积层（Conv）、Batch Normalization 层、ReLU 激活层、Pooling 层、上采样层，Softmax 层。图中左半部分为编码（Encoder）阶段，右半部分为解码（Decoder）阶段

SegNet 的网络框架流程如下。

SegNet的网络框架流程

（1）编码阶段：把图像输入卷积神经网络进行向前传播，并在每次 Pooling 层中记录下 Pooling 特征的位置。

（2）解码阶段：对解码阶段得到的特征图进行上采样操作和卷积操作，将（1）中编码得到的特征转换为图像标签的过程。

（3）分类阶段：逐像素进行 Softmax 分类，求得每个像素所属类别标签。

上采样层

以 Max Pooling 为例，如图 5-20（a）所示，SegNet 在 Pooling 下采样操作阶段不仅提取窗口内最大的像素值特征，而且记录了该最大像素特征在原图的位置。在后续上采样层的 Unpooling 操作，如图 5-20（b）所示，利用 Pooling 记录的最大像素特征位置，将特征映射回原图中对应的位置，其余像素用零填充。这样做的好处是使得在 Pooling 过程丢失的位置信息在解码阶段得到恢复。

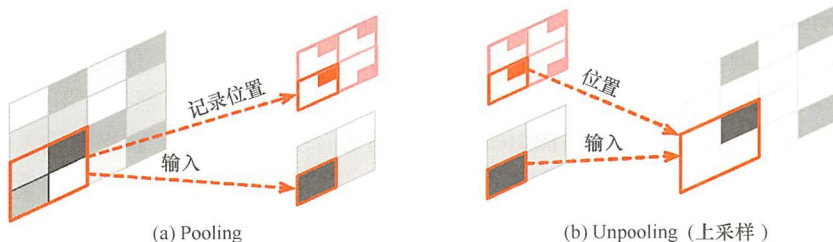


图 5-20 Pooling 与 Unpooling 操作。(a) 为 Pooling 操作，目标是对输入的图像进行降采样，并记录降采样的位置；(b) 为 Unpooling 操作，目标是对图像进行上采样操作恢复图像大小

### SegNet 小结

使用编解码思想对图像进行语义分割的框架还有 DeconvNet (Noh, 2015) 等，其网络模型结构的对称性不仅易于理解，而且使得网络训练更加方便（同样能实现端到端的训练）。

## 5.2.4 DeepLab网络

DeepLab 与 FCN 网络模型可以说是深度学习进行图像语义分割的经典之作，2016 年后获得较高 mIOU 的图像语义分割方法大部分是基于 DeepLab 和 FCN 的思想进行改进的。因此 DeepLab 的发表奠定了近年来图像语义分割的通用框架：

- 微调 FCN 网络；
- 使用 CRF 或者 MRF 进行像素分割；
- Softmax 输出分割图。

从图 5-21 所示的 DeepLab 的效果图可以看出，在微调 FCN 网络模型的基础上，加入了全连接 CRF 算法后，最终的图像语义分割精度进一步提升，在 CRF 迭代次数为 10 时，获得最好的分割效果。

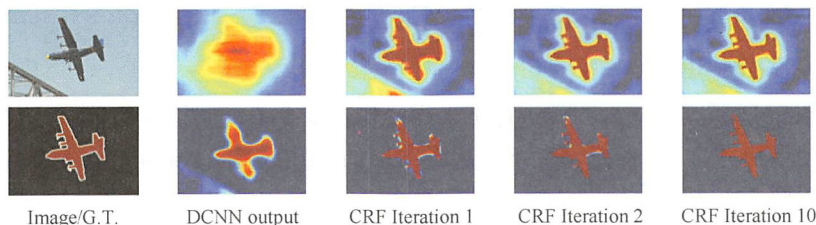


图 5-21 DeepLab 的分割效果图 (Softmax Output) 和得分图 (Score Map)。图中上半部分为经过 DeepLab 网络提取到的得分图，下半部分为经过 Softmax 层预测的图像分割效果图。

其中，CRF Iteration 1 表示 1 次迭代的全连接 CRF 结果

DeepLab 网络在卷积神经网络模型的基础上提出了空洞卷积层（Atrous 卷积层）和金字塔型空洞池化层（Atrous Spatial Pyramid Pooling, ASPP），使得卷积神经网络模型在不降低图像空间维度的前提下增大了卷积层的感知区域。然后使用 FCN 的思想获得得分图（FCN 中称为热图 Heat Map）后进行上采样操作，获得与输入图像相同尺寸的图像。接着引入全连接条件随机场（Full Connected CRF）进行亚像素级别的分割，最终输出分割图。

DeepLab 网络框架流程如下，对应的架构图如图 5-22 所示。

### DeepLab 的网络框架流程

- （1）对 FCN 网络进行微调修改，把部分卷积层和 Pooling 层换成 Atrous 卷积层。
- （2）对经过 FCN 网络产生的得分图进行上采样操作，获得与输入图像相同尺寸的特征图像。
- （3）使用全连接 CRF 计算上采样后图像的得分图。
- （4）经过 Softmax 层获得分割图。

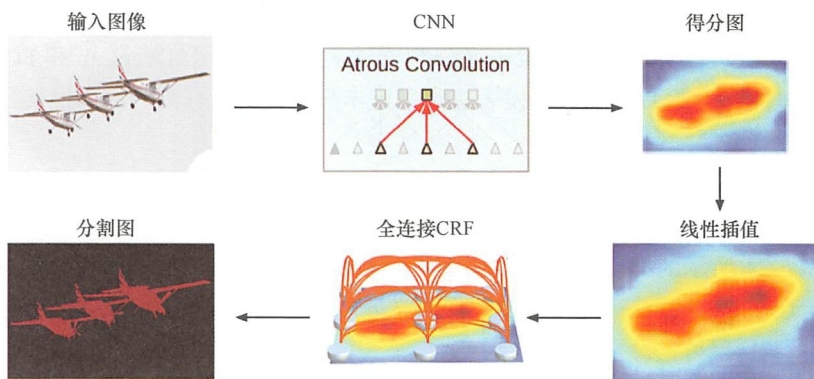


图 5-22 DeepLab 架构图

### Atrous 卷积层

Pooling 层通过对输入图像进行下采样操作，减少下一层输入的神经元数（即减少输出图像尺寸），目的是增大下一层卷积层的感知区域。Pooling 层的缺点是降低了输出图像的分辨率，丢失了输入图像中的部分特征，而 Atrous 卷积层则是在不降低空间维度的前提下增大相应卷积层的感知区域。

为了保证输出的尺寸不会太少，我们可以减少 Pooling 层。因为减少 Pooling 层相当于减少图像缩放的次数，但是这样的操作方式会直接改变 FCN 网络的结构，而

不能基于训练好的 FCN 网络模型参数进行微调。因此，Deeplab 在这里采用了一个非常优雅的策略：将 Pooling 层的步长改为 1 ( $stride=1$ ,  $padding=1$ )。这样 Pooling 层后图像尺寸并没有减小，并且保留了 Pooling 层提取的特性。接着使用 Atrous 卷积层提取更加密集的高维特征，同时增大感知区域。

图 5-23 所示为对一维数据进行卷积操作，圆圈代表输入数据，方框代表输出数据。假设卷积核大小为 3，步长为 1，普通的卷积操作提取稀疏特征是对输入的数据进行卷积，即对输入中连续的 3 个数据进行卷积操作，合并为一个数据作为输出，最后如图 5-23 (a) 中实线所示。而 Atrous 卷积操作则是如图 5-23 (b) 中实线所示，设置  $rate$  参数为 2，也就是将输入中的 3 个数据相隔开，2 个进行卷积操作，从而实现密集特征的提取，接受更大的感知区域。

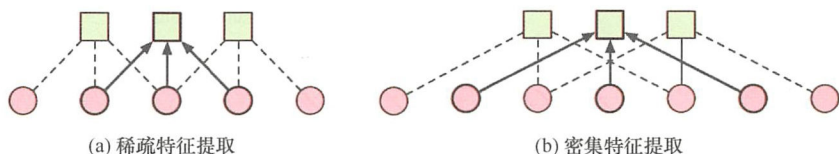


图 5-23 Atrous Convolution 方法。(a) 为普通卷积方式，其中  $kernel=3$ 、 $stride=1$ 、 $padding=1$ ；  
(b) 为 Atrous 卷积方式，其中  $kernel=3$ 、 $stride=1$ 、 $padding=2$ 、 $rate=2$

在 FCN 或者 SegNet 中，我们需要对原图进行一次下采样操作把原图缩放为  $1/4$ ，然后对特征图进行  $3 \times 3$  的卷积操作，最后通过上采样操作恢复原图大小。而 DeepLab 引入 Atrous 卷积层后，只需要对原图进行一次  $3 \times 3$  的 Atrous 卷积操作，得到同样大小的得分图更加精细，拥有更丰富的特征。

### ASPP 层

DeepLab 还提出了金字塔型的空洞池化层 (Atrous Spatial Pyramid Pooling, ASPP)。ASPP 层的原理是对输入特征图的同一个像素位置中心采用图像金字塔的方式，多次对该像素点进行不同参数 Atrous 卷积操作，从而在同一特征图上使用不同层次感知区域进行卷积，获得不同的图像特征。

图 5-24 所示为使用了 4 个不同的  $rate$  参数，实现 4 层金字塔型的空洞池化层。假设该层网络的输入特征图大小为  $64 \times 64$ ，ASPP 使用了 4 次不同  $rate$  参数，进行相同卷积核的卷积操作，得到的特征图作为下一层的输入。

### DeepLab 小结

DeepLab 提出了 3 个创新点。

- Atrous 卷积操作使输出的结果更加精细。
- 引入了 Atrous SPP 层，兼顾像素间的局部关系。



- 引入概率图模型中的条件随机场（CRF）算法，使分割图更加精细，也因此奠定了近年来图像语义分割的通用框架。（概率图模型中的内容不在本章的论述范围内，有兴趣的读者可以深入阅读（Daphne Koller et al.）的《概率图模型：原理与技术》。）

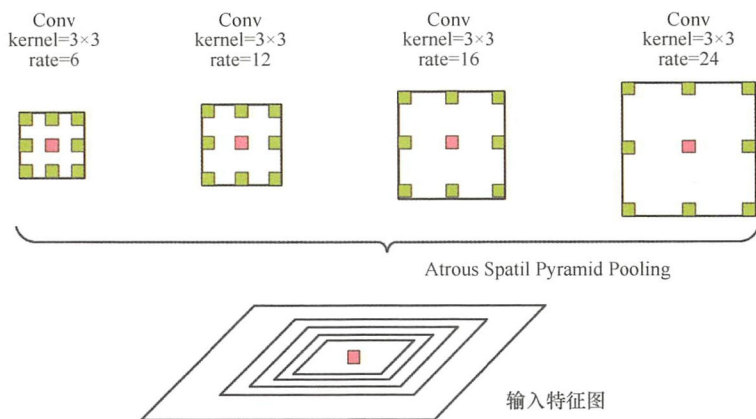


图 5-24 Atrous SPP 层。分别使用了 rate=6、12、16、24 这 4 个不同的参数对特征图进行 Atrous 卷积操作，最后对 4 个得到的特征矩阵进行求和

## 5.3 示例1：NMS确定候选框

在图像检测领域中，非极大值抑制算法（Non-Max Suppression, NMS）应用十分广泛。其中在目标检测的 YOLO 架构中最后一个步骤，需要对  $S \times S \times (B \times 5 + C)$  的特征经过 NMS 算法求得最终的目标窗口。

简单来说，NMS 算法的目的是消除多余的 Bounding-box，找到最佳的物体检测的位置。如图 5-25 (b) 所示，有 4 个红色的 Bounding-box，其所属类别均为蒲公英，其 Bounding-box 的置信度分别是  $[0.5, 0.7, 0.6, 0.7]$ 。那到底哪一个 Bounding-box 才最有可能是蒲公英呢？NMS 算法就是在这 4 个 Bounding-box 当中选出置信度最高且最有可能是蒲公英的 Bounding-box。

在【代码清单 5-1】中使用 numpy 定义 4 个窗口，并且设置其置信度分别是  $[0.5, 0.7, 0.6, 0.7]$ 。因此 dets 为  $4 \times 5$  的矩阵，最后一列为置信度值，其余分别是 Bounding-box 的左上角  $(x1, y2)$  和右下角  $(x2, y2)$  对应的坐标。

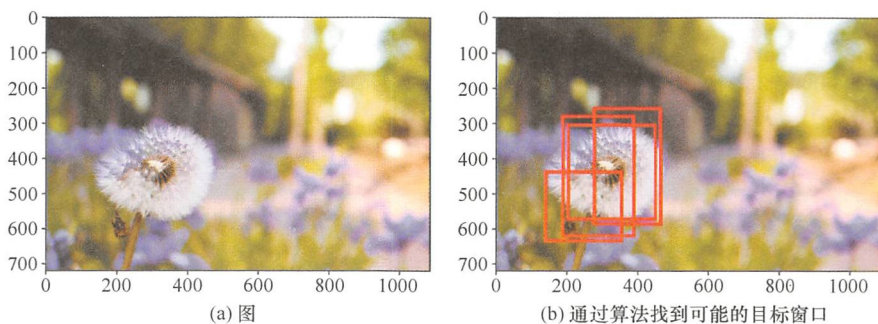


图 5-25 经过图像目标检测算法后得到属于蒲公英的 Bounding-box

### 【代码清单 5-1】 Bounding-box 定义

```
import numpy as np

# 定义 Bounding-boxes
dets = np.array([[204, 102, 358, 250, 0.5],
                 [257, 118, 380, 250, 0.7],
                 [280, 135, 400, 250, 0.6],
                 [255, 118, 360, 235, 0.7]])
```

### NMS算法的核心流程

- (1) 首先将 Bounding-box 按照置信度从大到小进行排列。
- (2) 把置信度最高的 Bounding-box 作为本次循环目标，计算余下的 Bounding-box 与该 Bounding-box 之间的交叉区域面积大小。
- (3) 如果交叉区域面积占比大于设定的阈值，则在剩下的 Bounding-box 中去除该 Bounding-box，否则存储该 Bounding-box。
- (4) 把置信度第二高的 Bounding-box 作为目标，重复步骤 (2) 和 (3)，直至遍历所有的 Bounding-box。

如【代码清单 5-2】所示，`scores = dets[:, 4]` 用于存储 Bounding-box 的置信度，通过使用 numpy 的内置函数 `argsort` 函数对 `scores` 向量进行排列。`(xx1, yy1)` 为交叉区域的左上角，`(xx2, yy2)` 为交叉区域的右下角，通过 `x1[order[1:]]` 直接求得当前 Bounding-box 与剩下所有 Bounding-box 的交叉窗口位置，`ovr` 则是交叉区域所占非交叉区域的大小比例。

### 【代码清单 5-2】 NMS 核心算法

```
# 设定过滤交叉面积的阈值
thresh = 0.3
```

```
def nms(dets, thresh):
    """ 非极大值抑制算法 """
    x1 = dets[:, 0]
    y1 = dets[:, 1]
    x2 = dets[:, 2]
    y2 = dets[:, 3]
    scores = dets[:, 4]

    areas = (x2 - x1 + 1) * (y2 - y1 + 1) # 计算每个 Bounding-box 的面积, +1 避免计算为 0
    order = scores.argsort()[::-1]        # 1) Bounding-box 的置信度排序
    keep = []                             # 用于保存最终获得的 Bounding-box

    # 遍历每一个 Bounding-box
    while order.size > 0:
        i = order[0]                      # 置信度最高的 Bounding-box 的 index
        keep.append(i)                    # 添加本次置信度最高的 Bounding-box 的 index

        # 选择大于 x1, y1 和小于 x2, y2 的区域
        xx1 = np.maximum(x1[i], x1[order[1:]])
        yy1 = np.maximum(y1[i], y1[order[1:]])
        xx2 = np.minimum(x2[i], x2[order[1:]])
        yy2 = np.minimum(y2[i], y2[order[1:]])

        # 2) 当前 Bounding-box 和其他剩下 Bounding-box 之间交叉区域的面积
        w = np.maximum(0.0, xx2 - xx1 + 1)
        h = np.maximum(0.0, yy2 - yy1 + 1)
        inter = w * h

        # 3) 交叉区域面积 / (Bounding-box + 某区域面积 - 交叉区域面积)
        ovr = inter / (areas[i] + areas[order[1:]] - inter)

        # 4) 保留交集小于一定阈值的 boundingbox
        inds = np.where(ovr <= thresh)[0]
        order = order[inds + 1]

    return keep

>>> Bounding-box = nms(dets, thresh)
>>> plt_Bounding-box(Bounding-box)
```

最后调用 `nms` 函数, 传入 `dets` 矩阵和 `thresh` 阈值, 并把最终结果传入 `plt_Bounding-box` 函数用于显示 `nms` 抑制后的 `Bounding-box` 结果 (如图 5-26 所示)。

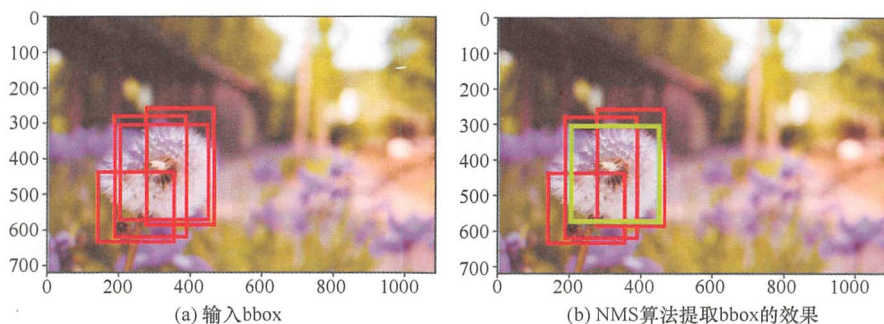


图 5-26 NMS 提取候选框结果。从 (b) 中可以看出, 中间绿色的 Bounding-box 为经过 NMS 算法求得的最终蒲公英位置

## 5.4 示例2: SS算法提取候选框

(JRR Uijlings et al.) 在 2013 年发表了选择性搜索 (SS) 算法, 本章中的 R-CNN、SPP-Net 等诸多基于深度学习的目标识别框架在提取候选框的阶段均使用了选择性搜索 (SS) 算法。

在本例中, 我们将会使用 Python 对输入的图像进行操作, 利用选择性搜索 (SS) 算法提取其目标图像的候选框。选择选择性搜索 (SS) 算法作为本章案例的原因在于选择性搜索 (SS) 算法中设计了诸多图像处理的基本知识 (如特征提取、图像直方图计算、图像分割等), 本例将会为我们进入深度机器视觉领域打下扎实的基础。

### 5.4.1 图像复杂度

一幅图像中可能会包含非常丰富的信息, 例如图像中不同的物体可以有不同的形状、尺寸、颜色、纹理等特征。假设想要从复杂的图像背景中识别出各种物体, 在不使用深度学习的知识的情况下, 我们会使用什么样的图像检测算法去做目标识别呢?

下面对图像的复杂度进行分析, 如图 5-27 所示。图 5-27 (a) 中表现了图像可以由层次划分, 图中的场景为饭桌, 饭桌上有各式各样的碗、瓶、勺子等餐具, 那么应该识别出图像中的饭桌, 还是饭桌上的餐具呢? 图像中不同的物体有一定的层次关系, 同一个物体也有不同的尺寸和形状大小。图 5-27 (b) 中表现了图像可以由颜色划分, 该图中包含了 2 只小猫, 算法可以通过纹理来识别出图中有猫, 但是想要区别是多少只猫, 可能就要靠颜色去划分。图 5-27 (c) 中表现了图像可以由纹理划分, 该图中的变色龙与周围的颜色很相近, 因此不能够用基于颜色的划分方法,



只能通过纹理去区别树叶和变色龙。图 5-27 (d) 中表现了图像可以由轮廓划分, 该图中有一辆跑车, 算法可以对跑车进行轮廓检测。



图 5-27 图像划分的多样性: (a) 中表现了图像可以通过层次划分; (b) 中表现了图像可以通过颜色划分; (c) 中表现了图像可以通过纹理划分; (d) 中表现了图像可以通过轮廓划分

图 5-27 中的不同图片具有不同的提取特征方式, 仅仅通过一种算法很难对图像中的物体进行判别的。这时我们就应该充分考虑图像中的多样性, 例如使用颜色 (color)、纹理 (texture)、大小 (size)、填充区域 (region)、层次 (hierarchical) 等多方面的特性对图像进行区分。而选择性搜索 (SS) 算法就是充分考虑图像中的多样性, 找出图像中可能是物体的区域。

## 5.4.2 算法核心思想

选择性搜索 (SS) 算法的核心是分层分组搜索 (Selective Search by Hierarchical Grouping), 分为如下 7 个步骤。

### 选择性搜索 (SS) 算法流程

- (1) 使用基于图论的快速图像分割 (Felzenszwalb et al., 2013) 作为底层理论支撑, 获得原始分割区域  $R = \{r_1, \dots, r_2\}$ 。
- (2) 初始化相似度集合  $S$ , 用于存储两两相邻区域的信息。
- (3) 计算原始分割区域中相邻区域之间的相似度, 将其添加到集合  $S$  中。
- (4) 遍历相似度集合  $S$ , 从中找出相似度最大的两个区域  $r_i$  和  $r_j$ , 并把  $r_i$  和  $r_j$  合并成为一个新的区域  $r_i$ 。
- (5) 从集合  $S$  中去掉  $r_i$  和  $r_j$  的信息。
- (6) 继续遍历  $r_i$  与其相邻区域的相似度, 将其结果添加的到相似度集合  $S$  中, 同时将新区域  $r_i$  添加到区域集合  $R$  中。
- (7) 最后区域集合  $R$  就是目标 Bounding-boxes。

选择性搜索 (SS) 算法中区域合并的方式是有层次的 (Hierarchical), 类似于哈

夫曼树的构造过程，因此称为分层分组搜索。选择性搜索（SS）算法流程对应的具体伪代码如下。

选择性搜索（SS）算法的 Hierarchical Grouping 伪代码

输入: image with colour

输出: Set of object location hypotheses

Obtain initial regions  $R = \{r_1, \dots, r_2\}$  using Efficient Graph-Based Image Segmentation (1)

Initialse similarity set  $S = \emptyset$  (2)

foreach Neighbouring region pair  $s(ri, rj)$  do

Calculate similarity  $s(ri, rj)$  (3)

$S = S \cup s(ri, rj)$

while  $S \neq \emptyset$  do

get highest similarity  $s(ri, rj) = \max(S)$  (4)

Merge corresponding regions  $rt = ri \cup rj$

Remove similarities regarding  $r_i; S = S \setminus s(r_i, r_*)$  (5)

Remove similarities regarding  $r_j; S = S \setminus s(r_*, r_j)$

Calculate similarity set  $S_t$  between  $r_t$  and its neighbours (6)

$S = S \cup S_t$

$S = S \cup r_t$

Extract object location Bounding-boxes from all regions in  $R$  (7)

选择性搜索（SS）算法代码很简单，重点在区域相似度的计算上。首先对输入的图像进行 Felzenszwalb 分割后得到不同分割窗口，然后计算这些分割窗口之间的相似度，将得分高的窗口合并为新的窗口，最终得到的窗口就是目标候选框。

在【代码清单 5-3】的 selective\_search() 函数中有一处比较难懂，就是从集合 S 中找出相似度最大的两个区域 ri 和 rj 的索引 i 和 j。sorted 函数是对数组 list 进行操作，而 S 集合是字典形式存储，因此需要对 S 字典进行数组转换 list(S.items())。其中，sorted(list, key=lambda a: a[1]) 是对转换为 list 的 S 进行排列，排列方式按照 key 中的 lambda 表达式，将 S 中单个单元的第二个元素 a[1]（第一个元素是 a[0]）按照从小到大的顺序排列。

【代码清单 5-3】Selective Search 流程代码

```
def selective_search (im_orig):  
    """ Selective Search 流程代码 """
```

```

# 图像分割算法实现函数 Efficient Graph-Based Image Segmentation, Felzenszwalb
img = graph_segment(im_orig)

R = extract_regions(img)          # 根据 Felzenszwalb 分割图获得分割区域
imsize = img.shape[0] * img.shape[1] # 获得原始图像大小
neighbours = extract_neighbours(R)  # 计算所有相邻区域的相似度

S = {}                             # 相似度集合 S 初始化

# 遍历相邻区域, 把其相似度信息存储在 S 集合中
for (ai, ar), (bi, br) in neighbours:
    S[(ai, bi)] = calc_sim(ar, br, imsize)

# SS 算法核心分层搜索的实现 hierarchal search
while S != {}:
    # 从 S 中找出相似度最大的两个区域 ri 和 rj 的索引 i 和 j
    highest = sorted(list(S.items()), key=lambda a: a[1])[-1]
    i, j = highest[0]

    # 把 ri 和 rj 合并成为一个新的区域 rt
    t = max(R.keys()) + 1.0
    R[t] = _merge_regions(R[i], R[j])

    # 从集合 S 中去掉 ri 和 rj 的信息
    key_to_delete = []
    for k, v in S.items():
        if (i in k) or (j in k):
            del S[k]
    for k in key_to_delete:
        del S[k]

    # 计算 rt 与其相邻区域的相似度
    for k in filter(lambda a: a != (i, j), key_to_delete):
        n = k[1] if k[0] in (i, j) else k[0]
        S[(t, n)] = calc_sim(R[t], R[n], imsize)

# 获取每个区域的 Bounding-boxes 信息
regions = []
for k, r in list(R.items()):
    regions.append({
        'rect': (r['min_x'], r['min_y'],
                  r['max_x'] - r['min_x'],
                  r['max_y'] - r['min_y']),
        'size': r['size'],
    })

```

```

        'labels': r['labels'] })

# 返回最后获得的区域
return regions

# Selective Search 函数细节展示
>>> S = {(1201.0, 492.0): 4.142, (494.0, 1211.0): 4.329, ...}

# 把 S 字典转换成为列表
>>> list(S.items())
>>> [(1201.0, 492.0), 4.142], ((494.0, 1211.0), 4.329), ...]
```

另外, `generate_segments` 是选择性搜索 (SS) 算法的基础, 作用于获得 Felzenszwalb 分割图, 如图 5-28 (b) 所示, 并把分割后的信息作为原始图像的第四个通道进行存储。如原始输入图像每个像素存储 `[r,g,b]` 这 3 个值, 经过 Felzenszwalb 分割函数处理后每个像素存储 `[r,g,b,Felzenszwalb]` 4 个值。



(a) 原始输入图像



(b) Felzenszwalb 分割算法

图 5-28 图像分割算法 Efficient Graph-Based Image Segmentation, Felzenszwalb

【代码清单 5-4】中输入的应为彩色图像, 因此加入 `assert` 断言进行判断。当检测到图像矩阵不是三通道时, 退出程序并输出错误信息。在 Felzenszwalb 分割算法的输入中, `scale` 是分割参数, 数值越小, 分割得越精细。`sigma` 是分割图像前对图像进行高斯平滑的参数, `min_size` 是分割的最小单元, `min_size` 一般设置在 10 ~ 100 之间都是合理的 (关于 Felzenszwalb 图像分割算法的具体内容, 有兴趣的读者可以进一步阅读相关文献)。

#### 【代码清单 5-4】Felzenszwalb 图像分割算法

```

from skimage import segmentation

def generate_segments(im_orig, scale=150, sigma=0.85, min_size=50):
    assert im_orig.shape[2] == 3, "输入应该是彩色图片"

    im_mask = segmentation.felzenszwalb(
        util.img_as_float(im_orig), scale=scale, sigma=sigma, min_size=min_size)
```



```

# 将掩码通道作为第四通道合并到图像
im_mask_ = np.zeros(im_orig.shape[:2])[:, :, np.newaxis] # (424, 640, 1)
im_orig = np.append(im_orig, im_mask_, axis=2) # (424, 640, 4)
im_orig[:, :, 3] = im_mask

return im_orig

```

分割完图像之后，需要提取分割区域。在【代码清单 5-5】中，`extract_regions()` 函数根据 Felzenszwalb 分割图像的每一个像素位置进行计算，属于同一分割区域的，则标记第一个出现的像素位置和最后一个出现的像素位置作为一个 Bounding-box。

【代码清单 5-5】根据 Felzenszwalb 分割图获得分割区域

```

def extract_regions(img):
    R = {}

    for y, i in enumerate(img): # 行遍历
        for x, (r, g, b, l) in enumerate(i): # 列遍历
            # 初始化新的区域
            if l not in R:
                R[l] = {"min_x": 0xffff, "min_y": 0xffff,
                        "max_x": 0.0000, "max_y": 0.0000,
                        "labels": [l]}

            # Bounding-box
            if R[l]["min_x"] > x: R[l]["min_x"] = x
            if R[l]["min_y"] > y: R[l]["min_y"] = y
            if R[l]["max_x"] < x: R[l]["max_x"] = x
            if R[l]["max_y"] < y: R[l]["max_y"] = y

    return R

```

到目前为止，我们已经大致了解了选择性搜索（SS）算法的工作原理。其中使用的区域合并算法的原理很简单，那么其真实效果又如何呢？作者在进行 ADAS 项目中的前方碰撞预警系统（Forward Collision Warning, FCW）时，使用非深度学习来检测前车正是基于该区域合并算法（改版）。从 6000 多张不同场景的图像中提取了约 5 万个候选框，然后通过该区域合并算法（改版）稳定地实现了前车检测，在 NVIDIA TX1 达到 100fps，mAP 为 97.6%。

### 5.4.3 区域相似度计算

实际上，选择性搜索使用了两大方面的多样化策略：

- 颜色空间多样化;
- 相似多样化。

其中, 颜色空间多样化采用了 Felzenszwalb 基于图论的图像分割方法, 产生原始区域。另外在相邻区域中, 根据相似度合并分割区域, 其中相似度的计算包括颜色、纹理、大小、填充 4 个方面。

### 1. 颜色相似度

$C_i = \{c_i^1, \dots, c_i^n\}$  为区域  $r_i$  对应的颜色直方图向量。如【代码清单 5-6】假设直方图的 bins 为 25, 那么图像区域中每个通道可以获得一个 25 维的向量, 三通道的图像区域可以获得  $25 \times 3 = 75$  维的向量 (即  $n=75$ )。区域间颜色相似度计算公式为:

$$S_{\text{colour}}(r_i, r_j) = \sum_{k=1}^n \min(c_i^k, c_j^k) \quad (5-14)$$

式 (5-14) 表明, 对两个区域颜色直方图中最少出现颜色的统计量求和, 两区域颜色越相似, 求和统计量就越大。

#### 【代码清单 5-6】计算颜色相似度

```
def calc_colour_hist(img):
    BINS = 25
    hist = np.array([])

    for colour_channel in (0, 1, 2):
        c = img[:, colour_channel]
        hist = np.concatenate([hist] + [np.histogram(c, BINS, (0.0, 255.0))[0]])
    hist = hist / len(img) # L1 normalize

    return hist

def sim_colour(r1, r2):
    """ 计算相邻区域的颜色相似度 """
    return sum([min(a, b) for a, b in zip(r1["hist_c"], r2["hist_c"])])
```

#### 颜色直方图

图像颜色直方图是在图像检索系统中被广泛采用的颜色特征, 其描述的是不同像素值在整幅图像中所占的比例, 可以反映图像颜色的统计分布和基本色调。如图 5-29 下半部分所示为单通道的颜色直方图, 这里使用了 25 个 bins, 每个 bins 统计其所在区域的颜色数量。

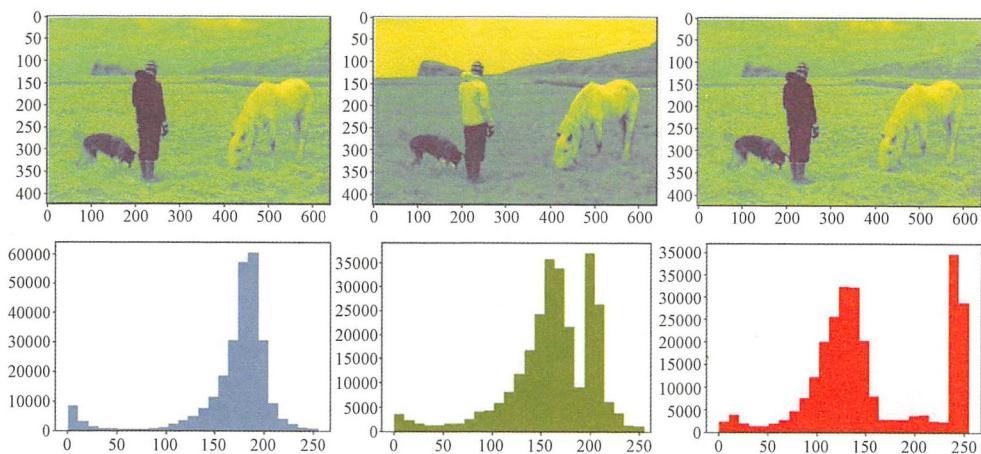


图 5-29 颜色直方图，其中颜色直方图的 bins 为 25。图中上半部分为原图的 (R, G, B) 三通道的灰度图，对应下图各自通道的颜色直方图，可以看出每个通道所代表的颜色统计量都是不同的

## 2. 纹理相似度

JRR Uijlings 在论文中使用 SIFT 特征，本例中采用局部二值模式 (Local Binary Pattern, LBP) 算法作为代替，计算其图像中的特征。

如【代码清单 5-7】所示，具体做法是计算每个颜色通道的 8 个不同方向的 LBP 特征的值，每个通道每个颜色获取 10 个 bins 的直方图。这样就可以获取到一个  $n=10 \times 8 \times 3$  维的向量  $T_i = \{t_i^1, \dots, t_i^n\}$ ，区域之间纹理相似度的计算方式和颜色相似度的计算方式类似，合并之后新区域的纹理特征的计算方式和颜色特征的计算相同。

$$S_{\text{texture}}(r_i, r_j) = \sum_{k=1}^n \min(t_i^k, t_j^k) \quad (5-15)$$

### 【代码清单 5-7】计算纹理相似度

```
from skimage.feature import local_binary_pattern as LBP

def calc_texture_gradient(img):
    # 建立新的 numpy 矩阵存储纹理 texture
    im_texture = np.zeros(img.shape[:3])

    # 逐个通道提取其 LBP 的特征
    for colour_channel in (0, 1, 2):
        im_texture[:, :, colour_channel] = LBP(img[:, :, colour_channel], 8, 1.0)

    return im_texture
```

```
def sim_texture(r1, r2):
    """ 计算相邻区域的纹理相似度 """
    return sum([min(a, b) for a, b in zip(r1["hist_t"], r2["hist_t"])])
```

### 局部二值模式 (Local Binary Pattern, LBP)

LBP 以邻域中心像素为阈值，将相邻的 8 个像素的灰度值与中心像素值进行比较，若周围像素值大于中心像素值，则该像素点的位置被标记为 1，否则为 0。这样， $3 \times 3$  邻域内的 8 个点经比较后可产生 8 位二进制数，将二进制转换成为十进制数即得到该邻域中心像素点的 LBP 值，并用这个值来反映该区域的纹理信息（具体如图 5-30 所示）。

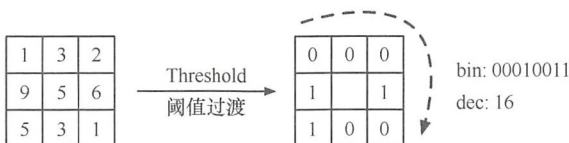


图 5-30 LBP 示例图，将一个  $3 \times 3$  的邻域内的 8 个点与中心像素值进行比较，得到一个 8 位的二进制 00010011，该二进制的十进制值 16 就是 LBP 值

### 3. 大小相似度

这里的大小是指区域中包含像素点的个数，也就是区域的面积，目的是尽量让小的区域优先合并。如【代码清单 5-8】所示，其大小相似度计算公式如下：

$$S_{\text{size}}(r_i, r_j) = 1 - \frac{\text{size}(r_i) + \text{size}(r_j)}{\text{size}(\text{im})} \quad (5-16)$$

其中， $\text{size}(\text{im})$  为原始图像的面积。该式表明  $r_i$  和  $r_j$  两个区域占全图面积越大，其相似度越低，因此两个小的区域其相似度得分更高。

#### 【代码清单 5-8】计算大小相似度

```
def _sim_size(r1, r2, imsize):
    return 1.0 - (r1["size"] + r2["size"]) / imsize
```

### 4. 填充相似度

为了衡量  $r_i$  和  $r_j$  两个区域是否更加吻合， $\text{size}(BB_{ij})$  指包含  $r_i$  和  $r_j$  的最小区域 Bounding-box。其区域填充相似度公式为：

$$S_{\text{fill}}(r_i, r_j) = 1 - \frac{\text{size}(BB_{ij}) - \text{size}(r_i) - \text{size}(r_j)}{\text{size}(\text{im})} \quad (5-17)$$

式 (5-17) 表明  $r_i$  和  $r_j$  两个区域的重叠域越高，填充相似度得分越高。【代码清



单 5-9】中 bbsize 取 r1 和 r2 的最大坐标减去最小坐标，得到包含 r1 和 r2 的最小区域坐标。

#### 【代码清单 5-9】计算填充相似度

```
def _sim_fill(r1, r2, imsize):
    bbsize = (
        (max(r1["max_x"], r2["max_x"]) - min(r1["min_x"], r2["min_x"])) *
        (max(r1["max_y"], r2["max_y"]) - min(r1["min_y"], r2["min_y"])))
    return 1.0 - (bbsize - r1["size"] - r2["size"]) / imsize
```

### 5. 总体相似度

如【代码清单 5-10】所示，计算上面 4 个相似度，最后将 4 个相似度的得分乘以每个得分的权重  $\alpha$ ，其中  $\alpha \in (0,1)$ ，得到总体相似度：

$$s(r_i, r_j) = \alpha_1 S_{\text{colour}}(r_i, r_j) + \alpha_2 S_{\text{texture}}(r_i, r_j) + \alpha_3 S_{\text{size}}(r_i, r_j) + \alpha_4 S_{\text{fill}}(r_i, r_j)$$

#### 【代码清单 5-10】计算总体相似度

```
def sim_sum(r1, r2, imsize):
    weight = [1, 1, 1, 1]
    return (sim_colour(r1, r2) * weight[0] +
            sim_texture(r1, r2) * weight[1] +
            sim_size(r1, r2, imsize) * weight[2] +
            sim_fill(r1, r2, imsize) * weight[3])
```

通过上面的代码，我们已经实现了选择性搜索（SS）算法，最后实现 main() 总函数，用于读入原始图像和 Felzenszwalb 分割算法参数。在【代码清单 5-11】中，candidates 使用 set 结构体存储新的候选窗，Python 中的 set 为集合不允许元素重复，简单的代码就可以做到去重功能。另外对限制候选窗口的尺寸大小进行阈值限制，使用 candidates 变量存储剩余的候选区域框。得到候选区域框的 candidates 后，使用 matplotlib 的 pyplot 和 patches，ax 用来记录已经画了的候选框，plt 用于显示，最终效果如图 5-31 所示。

#### 【代码清单 5-11】算法主函数

```
def main():
    img = io.imread("person.jpg") # skimage.io 读入图片
    regions = selective_search(img, scale=500, sigma=0.9, min_size=10)

    candidates = set() # 过滤找到的区域
    for r in regions:
        if r['rect'] in candidates: continue
        if r['size'] < 100: continue
        candidates.add(r['rect'])
```

```
fig, ax = pyplot.subplots(ncols=1, nrows=1, figsize=(6, 6))
ax.imshow(img)
for x, y, w, h in candidates:
    rect = patches.Rectangle((x, y), w, h,
                             fill=False, edgecolor='red', linewidth=1)
    ax.add_patch(rect)

plt.show()
```

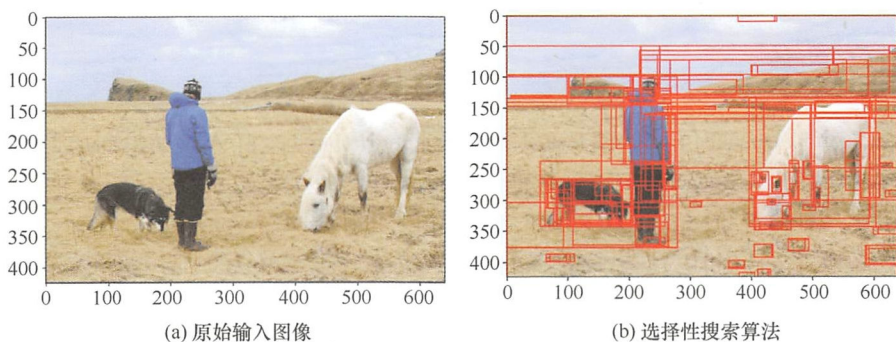


图 5-31 选择性搜索 (SS) 算法结果。(a) 为原始输入图, (b) 为最后通过选择性搜索 (SS) 算法的输出候选框

通过选择性搜索 (SS) 算法可以知道, 候选框已经包括了图中需要识别的物体 (狗、马、人)。其中部分候选框, 例如天空、小草堆都会被选出来, 后面如何进一步识别或抑制这些大量的候选区域框, 就要使用到本章图像目标检测中的深度学习方法了。

## 5.5 本章小结

在计算机视觉领域中, 基于深度学习的卷积神经网络模型是最直接有效的算法, 近年来包揽了各大涉及机器视觉比赛的前几名。本章通过分析近年来基于深度学习的图像检测框架, 从最初提出的 R-CNN 使用候选区域框 + CNN + SVM, 因为其运行时间慢、效率低, 从而衍生出 Fast R-CNN 和 Faster R-CNN, 后来有了 CNN + NMS 的 YOLO 和 SSD, 把分类和回归问题一起作为损失函数进行计算, 降低训练难度的同时, 也能够让图像检测实时工作。

图像检测的另外一个方向则是图像分割, FCN 首次提出全卷积的概念, 只需要

通过一次前馈网络即可逐个像素预测其分类，缺点是精度低、忽略像素间关系、运算有待提高。后来 DeepLab 提出 FCN+CRF，统一了图像检测的框架，并且更进一步地提高了精度，缺点是图像语义分割的运算速度还达不到实时的要求。

NMS 例子中得到每个候选框的置信度后，计算其 Bounding-Box 的置信度和重叠关系，对同一所属类别的 Bounding-Box 进行抑制，求得最终的 Bounding-Box。另外一个例子则是选择性搜索（SS）算法，使用基于图论的 Felzenszwalb 图像分割算法作为基础，并引入了多样化策略：通过对图像空间转换并计算相邻区域颜色相似度，提取图像特征计算纹理相识度。相信本章内容能让读者对图像处理有更深入的理解。

## 引用/参考

- [1] Girshick R, Donahue J, Darrell T, et al. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation[J]. Computer Science, 2014:580-587.
- [2] He K, Zhang X, Ren S, et al. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2014, 37(9):1904-1916.
- [3] Girshick R. Fast R-CNN[J]. Computer Science, 2015.
- [4] Ren S, Girshick R, Girshick R, et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2015, 39(6):1137-1149.
- [5] Redmon J, Divvala S, Girshick R, et al. You Only Look Once: Unified, Real-Time Object Detection[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2016:779-788.
- [6] Liu W, Anguelov D, Erhan D, et al. SSD: Single Shot MultiBox Detector[C]// European Conference on Computer Vision. Springer, Cham, 2016:21-37.
- [7] Long J, Shelhamer E, Darrell T. Fully convolutional networks for semantic segmentation[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2015:3431-3440.
- [8] Badrinarayanan V, Kendall A, Cipolla R. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2015, PP(99):1-1.
- [9] Chen L C, Papandreou G, Kokkinos I, et al. Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs[J]. Computer Science, 2014(4):357-361.
- [10] Koller D, Friedman N. Probabilistic Graphical Models: Principles and Techniques - Adaptive

- Computation and Machine Learning[M]. MIT Press, 2009.
- [11] Neubeck A, Gool L V. Efficient Non-Maximum Suppression[C]// International Conference on Pattern Recognition. IEEE Computer Society, 2006:850-855.
  - [12] Felzenszwalb P F, Huttenlocher D P. Efficient Graph-Based Image Segmentation[J]. International Journal of Computer Vision, 2004, 59(2):167-181.
  - [13] Uijlings J R, Sande K E, Gevers T, et al. Selective Search for Object Recognition[J]. International Journal of Computer Vision, 2013, 104(2):154-171.
  - [14] He K, Gkioxari G, Dollár P, et al. Mask R-CNN[J]. 2017.
  - [15] Li S Z. Markov Random Field Models in Computer Vision[C]// European Conference on Computer Vision. Springer Berlin Heidelberg, 1994:361-370.
  - [16] Gidaris S, Komodakis N. Object Detection via a Multi-region and Semantic Segmentation-Aware CNN Model[C]// IEEE International Conference on Computer Vision. IEEE Computer Society, 2015:1134-1142.
  - [17] Bell S, Zitnick C L, Bala K, et al. Inside-Outside Net: Detecting Objects in Context with Skip Pooling and Recurrent Neural Networks[C]// Computer Vision and Pattern Recognition. IEEE, 2016:2874-2883.
  - [18] Shrivastava A, Gupta A, Girshick R. Training Region-Based Object Detectors with Online Hard Example Mining[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2016:761-769.
  - [19] Chen L C, Papandreou G, Kokkinos I, et al. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2016, 40(4):834-848.
  - [20] Zheng S, Jayasumana S, Romera-Paredes B, et al. Conditional Random Fields as Recurrent Neural Networks[J]. 2015:1529-1537.
  - [21] Noh H, Hong S, Han B. Learning Deconvolution Network for Semantic Segmentation[C]// IEEE International Conference on Computer Vision. IEEE Computer Society, 2015:1520-1528.
  - [22] 王若辰. 基于深度学习的目标检测与分割算法研究 [D]. 北京工业大学, 2016.



---

# 第 6 章

---

## 卷积神经网络 进阶示例

本章主要内容：

- 全卷积网络图像语义分割
- 深度可视化卷积神经网络模型
- 卷积神经网络艺术绘画

通过前面对卷积神经网络的学习，我们掌握了卷积神经网络的基础知识，因为其网络结构模型能够高效地提取图像高维特征，因此卷积神经网络能够很好地处理图像分类任务。利用从图像中提取的高维特征，我们可以进行图像目标识别、图像语义分割等重要的任务。

在本章中，我们会对卷积神经网络进行深度剖析：从卷积神经网络模型开始，转换成全卷积网络（Fully Convolutional Networks, FCN）模型并对图像进行语义分割。接下来将会利用 FCN 模型结构和梯度上升算法，对卷积神经网络进行深度可视化分析，深入卷积神经网络模型内部并了解每一个卷积核的作用。在本章结束之前，将会利用卷积神经网络的特征提取风格特征，并将风格特征和图像内容特征融合进行艺术创造。

## 6.1 示例1: 全卷积网络图像语义分割

也许你无法想到，仅仅通过合理的损失函数加上简单的梯度下降算法，就能在图像中学习到物体的高维特征，在如此复杂和庞大的数据集中得到优秀的视觉预测模型。或许深度学习与实际的人工智能还有很远的距离，但是卷积神经网络毫无疑问地已经达到几年前任何人、任何算法都无法实现的效果。可是我们并不能仅满足于使用卷积神经网络的模型对图像进行分类的任务。

无论是高精度地图采集、无人驾驶领域，还是对车载终端采集到的视频图像进行语义分割，都十分重大的意义。只有进一步利用图像上的信息，才能够更精确地获得其真实世界的坐标，从而实现高级感知操作。全卷积网络（FCN）正是使用深度学习对图像进行语义分割的开山之作。在本节中，我们将会使用 VGG16 网络结构，并对其修改成 FCN，最后进行图像语义分割操作。

### 6.1.1 VGG连续小核卷积层

2014 年是卷积神经网络架构飞速发展的一年，这一年研究者们提出了两个重要的卷积神经网络框架——GoogleNet 和 VGGNet。其中 VGGNet 网络模型引入了一个重要思想：卷积层中使用  $3 \times 3$  卷积核，并连续多次  $3 \times 3$  卷积核的卷积层。例如：对原始图像进行  $3 \times 3$  卷积后，继续进行  $3 \times 3$  卷积，再继续进行  $3 \times 3$  的卷积，如此连续地使用小卷积核对图像进行多次卷积操作。

下面来分析为什么 VGGNet 网络模型需要使用如图 6-1 所示的小卷积核对图像进行卷积，并且还是连续的小卷积核。

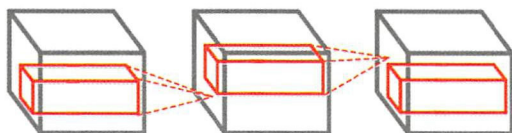


图 6-1 连续使用多个相同大小的小卷积核对输入的图像进行卷积操作  
(图中大的方框为输入卷积层的多层特征图，内部小方框为卷积核)

VGGNet 网络模型的设计原则与 AlexNet 网络模型相违背。AlexNet 网络模型相信浅层网络中较大的卷积核能够捕获图像中较大的特征。这些较大的特征在图像中有较多的相似性，可以更有效地进行权值共享。因此 AlexNet 网络模型的第一层卷积层使用  $11 \times 11$  大小的卷积核，并尽量在浅层网络避免使用小卷积核。

VGGNet 网络模型的观点刚好相反，在浅层网络连续使用多个大小为  $3 \times 3$  的小卷积核进行卷积操作。其声称该操作可以模仿大卷积核的特征提取结果，取得更好效果的同时，还能有效降低卷积神经网络的权重参数。

为什么连续多次小卷积核的卷积层组合，与大卷积核的卷积层具有相类似的感知区域？假设重复 3 次卷积核大小为  $3 \times 3$  的卷积层：第一个  $3 \times 3$  卷积层中的每个神经元都对输入数据有一个  $3 \times 3$  的感知区域，第二个  $3 \times 3$  卷积层上的神经元对第一个卷积层的输出继续进行  $3 \times 3$  的感知，也就是对最初输入数据产生了大小为  $6 \times 6$  的感知区域。同理，在第三个  $3 \times 3$  卷积层继续对第二个卷积层的输出进行  $3 \times 3$  的感知，最终相当于对原始输入的图像产生  $9 \times 9$  大小的感知区域。

连续多层小核卷积层与单层大卷积核的卷积层相比较，不仅能够产生相类似的感知区域，还带来了性能上的优化。

(1) 连续多层卷积的网络结构比单卷积层的网络结构能更有效地提取出图像中的高维特征。因为卷积神经网络中每一层都经过独立的训练，所提取的特征均不一样，层数越深，所提取特征维度越高。因此，连续多层卷积所提取到的特征比使用单层卷积提取的特征更加丰富。

(2) 大幅度地减少卷积神经网络中的权重参数。例如  $3 \times 3$  卷积核有 9 个权值参数，使用  $7 \times 7$  的卷积核权值参数高达 49 个，是  $3 \times 3$  卷积核的 5 倍。由于缺乏方法对网络中大量的权重参数进行归一化、约减等操作，因此训练大卷积核的卷积神经网络模型就变得非常困难，对硬件和时间都有着更多的需求。

(3) 有效减少边界特征的损失。假设卷积操作均使用 Same Padding，使得输入和输出特征矩阵大小相同。大卷积核在卷积前会在输入矩阵边缘填充更多的无效数据，导致卷积后产生大量的无效数据，影响特征提取的准确性。

综上所述，选择连续小卷积的卷积层组合比一个带有大卷积核的单层卷积层会取得更好的表现。这样可以使使用更少的参数，有效地表达出输入数据中的高维特征。



唯一的不足是在进行反向传播时，中间的卷积层的参数和求导参数可能会占用更多的内存，随着层数的加深，运算时间也会增加。

VGGNet 网络模型深信使用大的卷积核会浪费存储空间，减少卷积核能够大幅度减少参数并节省运算开销。虽然网络的层数增加了，但总体来说预测的时间和参数都减少了。正是这样独特的网络结构设计，使得 VGGNet 网络模型在 ImageNet 2014 比赛中斩获第一名。

## 6.1.2 VGG网络模型

图 6-2 所示为 VGGNet 的网络模型架构，其网络架构一共有 5 个 Block，每个 Block 都连续多次使用带有大小为  $3 \times 3$  卷积核的卷积层（图中连续出现大小相同的黑色方框）。该网络一共有 13 个卷积层、5 个 Pooling 层、2 个全连接层和 1 个输出层。

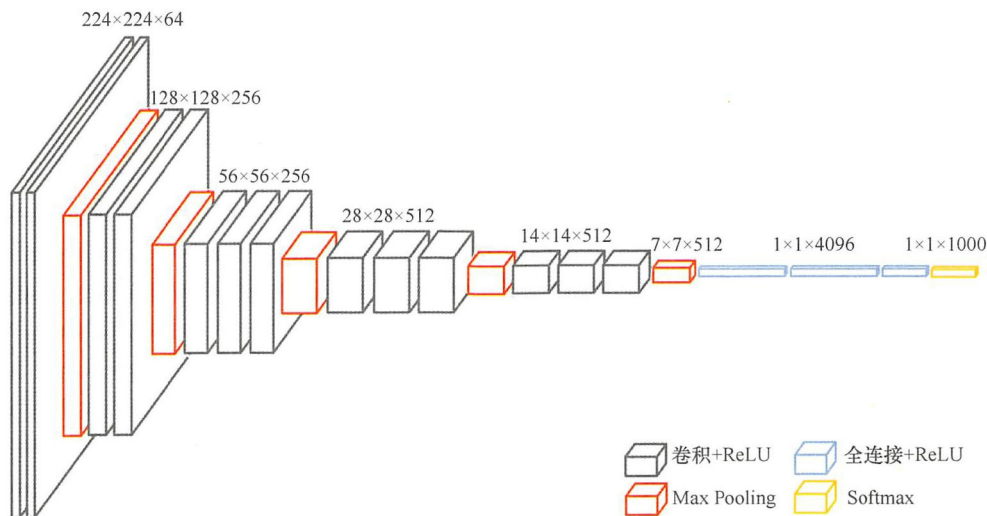


图 6-2 VGGNet 网络模型架构图。输入为  $224 \times 224$  的图像矩阵，输出为  $1 \times 1 \times 1000$  的分类概率

VGGNet 的网络模型架构详情如下。

- 输入层（Input）：输入为  $3 \times 224 \times 224$  或者  $1 \times 224 \times 224$  大小图像矩阵。
- Block1：连续两层  $\text{kernel}=64$ ， $\text{kernel size}=3 \times 3$  的卷积层，后接一个 Pooling 层。卷积层输出特征矩阵为  $[224, 224, 64]$ ，Pooling 层输出特征矩阵为  $[128, 128, 64]$ 。
- Block2：连续两层  $\text{kernel}=256$ ， $\text{kernel size}=3 \times 3$  的卷积层，后接一个 Pooling 层。卷积层输出特征矩阵为  $[128, 128, 256]$ ，Pooling 层输出特征矩阵为



[56,56,256]。

- Block3: 连续三层 kernel=256, kernel size=3×3 的卷积层, 后接一个 Pooling 层。卷积层输出特征矩阵为 [56,56,256], Pooling 层输出特征矩阵为 [28,28,256]。
- Block4: 连续三层 kernel=512, kernel size=3×3 的卷积层, 后接一个 Pooling 层。卷积层输出特征矩阵为 [28,28,512], Pooling 层输出特征矩阵为 [14,14,512]。
- Block5: 连续三层 kernel=512, kernel size=3×3 的卷积层, 后接一个 Pooling 层。卷积层输出特征矩阵为 [14,14,512], Pooling 层输出特征矩阵为 [7,7,512]。
- 全连接层 (FC1): 特征矩阵一维向量化为 25088 (7×7×512) 个神经元与 4096 个神经元进行全连接。
- 全连接层 (FC2): FC1 的 4096 个神经元与 FC2 中的 4096 个神经元进行全连接。
- 输出层 (Softmax): FC2 的 4096 个神经元与输出层的 1000 个分类进行全连接, 并用 Softmax 函数进行分类预测。

下面使用 Keras 来实现图 6-2 所示的 VGGNet 网络模型。在【代码清单 6-1】中, 定义图像大小为 [3,224,224] 作为输入, 后面开始 VGGNet 网络的 Block 结构, 每个 Block 连续多次使用带有 3×3 大小卷积核的卷积层, 最后加上 Max Pooling 层下采样输出特征图后输给下一个 Block。其中, 每次经过 Block 后输出的特征图为输入前的特征图尺寸的一半。

#### 【代码清单 6-1】VGGNet 网络模型定义

```
# 输入图像大小
input_shape = (224, 224, 3)
img_input = Input(shape=input_shape)

# Block1
x = Conv2D(64, 3, 3, activation='relu', border_mode='same', name='block1_conv1')(img_input)
x = Conv2D(64, 3, 3, activation='relu', border_mode='same', name='block1_conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block1_pool')(x)

# Block2
x = Conv2D(128, 3, 3, activation='relu', border_mode='same', name='block2_conv1')(x)
x = Conv2D(128, 3, 3, activation='relu', border_mode='same', name='block2_conv2')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block2_pool')(x)

# Block3
```

```

x = Conv2D(256, 3, 3, activation='relu', border_mode='same', name='block3_conv1')(x)
x = Conv2D(256, 3, 3, activation='relu', border_mode='same', name='block3_conv2')(x)
x = Conv2D(256, 3, 3, activation='relu', border_mode='same', name='block3_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block3_pool')(x)

# Block4
x = Conv2D(512, 3, 3, activation='relu', border_mode='same', name='block4_conv1')(x)
x = Conv2D(512, 3, 3, activation='relu', border_mode='same', name='block4_conv2')(x)
x = Conv2D(512, 3, 3, activation='relu', border_mode='same', name='block4_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block4_pool')(x)

# Block5
x = Conv2D(512, 3, 3, activation='relu', border_mode='same', name='block5_conv1')(x)
x = Conv2D(512, 3, 3, activation='relu', border_mode='same', name='block5_conv2')(x)
x = Conv2D(512, 3, 3, activation='relu', border_mode='same', name='block5_conv3')(x)
x = MaxPooling2D((2, 2), strides=(2, 2), name='block5_pool')(x)

x = Flatten(name='flatten')(x)
x = Dense(4096, activation='relu', name='fc1')(x)
x = Dense(4096, activation='relu', name='fc2')(x)
x = Dense(1000, activation='softmax', name='predictions')(x)

# 完成 vgg16 网络的定义
vgg16 = Model(img_input, x)
vgg16.load_weights(weights_path)

```

下面输出该 VGG16 网络模型的参数信息，如【代码清单 6-2】所示。从图中的信息可以看出，该 VGG16 网络模型一共使用了 23 层，每一层所对应的参数逐渐增多，并在全连接层之后网络的参数开始减少。（VGG19 则是在 Block3 到 Block5 中增加多一层卷积层，共增加 3 层卷积层）。

从 VGG16 网络模型的输出可以看出，该网络模型的输入图像尺寸为 [224,224]，一共有 5 个 Block，每个 Block 带有的 Pooling 层会把输入特征矩阵的尺寸空间缩减一半，因此在最后一层 Pooling 层得到的特征图尺寸为 [7,7]（这里没有包含网络特征的深度）。

#### 【代码清单 6-2】VGG16 网络模型

```
>>> vgg16.summary()
```

```
Model loaded.
```

Layer (type)	Output Shape	Param #
=====		

## 198 | 第6章 卷积神经网络进阶示例

input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312

```
predictions (Dense)          (None, 1000)          4097000
```

```
=====
```

```
Total params: 138,357,544
```

```
Trainable params: 138,357,544
```

```
Non-trainable params: 0
```

### 6.1.3 全卷积网络模型

如图 6-3 所示,普通的卷积神经网络分为两部分(卷积层和全连接层),对于输入  $(227,227,3)$  大小的图像,最后一层卷积后得到的特征为  $(13,13,256)$ 。然后把该特征进行展开作用于下一层的全连接层,经过两次全连接层之后输出该图像的分类概率。

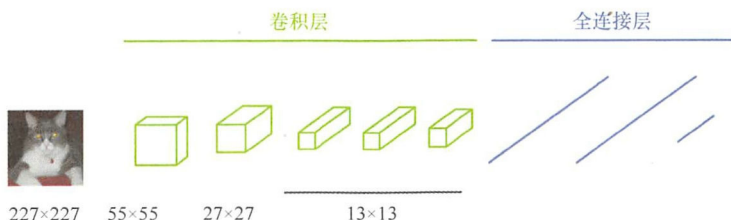


图 6-3 普通卷积神经网络 CNN 模型架构

FCN 网络模型的关键是把全连接层转换为卷积层。如图 6-4 所示,为了使得修改为 FCN 后的网络模型权重参数与卷积神经网络模型权重参数的数量保持一致,原卷积神经网络中 FC1 的全连接特征为 4096。FCN 进行全卷积修改为后 FC1 层的特征为  $(1,1,4096)$ ,这里只增加了网络层的特征深度,而没有改变总的特征数量。因此全连接层改为卷积层后,网络的模型权重参数结构仍然与图 6-3 中一致。后续两层(FC2 和输出层)同样适用。

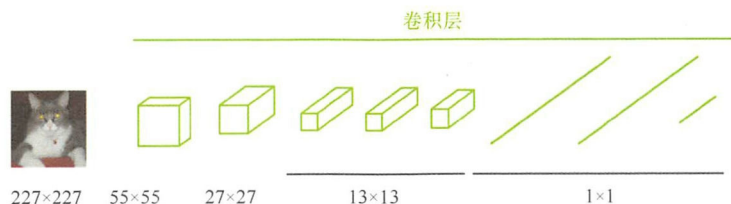


图 6-4 全卷积神经网络 (FCN) 模型架构

图 6-4 中 FCN 输入与图 6-3 中卷积神经网络的输入图像大小均为  $(227,227)$ ,它



## 200 | 第6章 卷积神经网络进阶示例

们在 FC1 层的连接神经元数均为 4096 个。如图 6-5 所示，修改 FCN 的图像输入大小，假设输入的图像大小为  $H \times W$ ，经过图 6-4 的 FCN 网络模型之后，对应 FC1、FC2、输出层的特征大小分别为  $(H/32, W/32, 4096)$ 、 $(H/32, W/32, 4096)$ 、 $(H/32, W/32, 1000)$ 。最后一层卷积后的特征矩阵作为热图（Heat Map），然后进行反卷积操作或者双线性插值法恢复原图大小用于像素级语义分割。

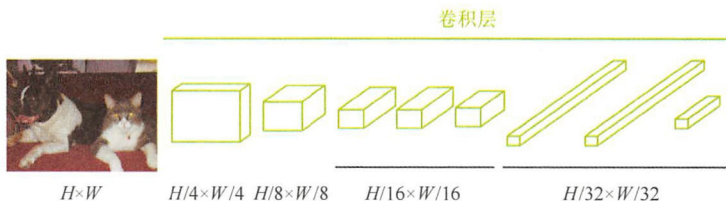


图 6-5 全卷积神经网络（FCN）模型架构

通过第 5 章图像语义分割中对 FCN 网络模型学习和上述的例图所示，我们知道了如何修改卷积神经网络模型得到 FCN 网络模型。接下来我们将会对 VGG16 模型进行 FCN 修改。

在【代码清单 6-3】的 FCN\_Vgg16() 函数中，定义最后三层全卷积层，原全连接层 FC1 改为 4096 个卷积核大小为  $7 \times 7$  的卷积层，原全连接层 FC2 改为 4096 个卷积核大小为  $1 \times 1$  的卷积层，原输出层改为 1000 个大小为  $1 \times 1$  的卷积层。

FC1 卷积核大小为  $7 \times 7$  的原因在于，从【代码清单 6-2】中的 VGG16 模型参数可以得知，FC1 的原输入神经元为 25088 个，输出为 4096 个，产生 102764544 个参数，只有当 FC1 的卷积核为  $7 \times 7$  时所产生的参数才能跟原 VGG16 模型一样。而后续 FC2 层的输入神经元为 4096 个，输出为 4096 个，产生 16781312 ( $4096 \times 4096 + 4096$ ) 个参数。除了必需的 4096 个偏置参数和  $4096 \times 4096$  的权重参数之外，并没有额外增加权重参数，因此后续卷积层的卷积核大小设定为  $1 \times 1$  即可。

### 【代码清单 6-3】将 VGG16 网络修改为 FCN 网络

```
def FCN32_Vgg16(Vgg16_Model):

    # 把全连接层转换成卷积层
    x = Conv2D(4096, (7, 7), activation='relu', padding='same', name='fc1')(x)
    x = Dropout(0.5)(x)
    x = Conv2D(4096, (1, 1), activation='relu', padding='same', name='fc2')(x)
    x = Dropout(0.5)(x)
    x = Conv2D(1000, (1, 1), activation='linear', padding='same', strides=(1, 1),
name='predictions')(x)

    # 线性插值进行上采样 32 倍操作
```

```
X = Upsample(size=(32, 32))(x)

# 加载原模型
model = Model(Vgg16_Model.input, X)
```

有了 FCN32\_Vgg16 函数模型的定义，接下来需要加载 VGG16 模型的权重参数，并修改其权重参数与 FCN32 网络模型的权重参数对应。在【代码清单 6-4】中，首先是通过遍历原模型的权重文件，获取原模型的每一层的名字，找到对应层的名字后使用 `get_weights()` 获取其网络的参数，然后通过 `reshape` 操作修改原网络模型的权重参数的排列方式（如果读者不了解原网络模型的权重参数排列方式，可以打印 `weight.shape` 内容进行查看）。在代码的最后把修改后的 FCN 网络模型权重参数保存起来，方便后续不需要重新对 VGG16 网络进行转换。

#### 【代码清单 6-4】接【代码清单 6-3】

```
weights_path = "fcn_vgg16_weights_tf_dim_ordering_tf_kernels.h5"

# 如果找不到 FCN 模型参数文件则创建一个
if os.path.isfile(weights_path) == False:

    # 把网络层的信息存储在 index 字典中，以网络层的名字作为索引
    flattened_layers = model.layers
    index = {}
    for layer in flattened_layers:
        if layer.name:
            index[layer.name] = layer

    # 遍历 vgg16 网络模型的权重参数文件
    for layer in vgg16.layers:
        # 获取该层网络的参数
        weights = layer.get_weights()

        # 开始模型参数重排列
        if layer.name == 'fc1':
            weights[0] = np.reshape(weights[0], (7, 7, 512, 4096))
        elif layer.name == 'fc2':
            weights[0] = np.reshape(weights[0], (1, 1, 4096, 4096))
        elif layer.name == 'predictions':
            layer.name = 'predictions'
            weights[0] = np.reshape(weights[0], (1, 1, 4096, 1000))
        if layer.name in index:
            index[layer.name].set_weights(weights)

    # 保存新的网络参数文件
    model.save_weights(weights_path)
```

```

        print('Successfully transformed!')
# 如果已经有 FCN 模型参数文件则直接加载
else:
    model.load_weights(weights_path, by_name=True)
    print('Already transformed!')

return model

```

接着，如【代码清单 6-5】所示，输出该 FCN 的网络模型。对比 VGG16 原模型的定义【代码清单 6-2】可以看出，将卷积神经网络的全连接层修改为 FCN 的全卷积层后，网络模型的参数数量没有发生任何变化，区别在于后面的全连接层变为卷积层，网络参数的排列方式从全连接层的二维变为卷积层的四维。

#### 【代码清单 6-5】输出 fcn32\_vgg16 模型

```

>>> fcn32_vgg16 = FCN32_Vgg16(Vgg16)
>>> print('Model loaded.')
>>> fcn_vgg16.summary()

```

```

Already transformed!
Model loaded.

```

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080

block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
fc1 (Conv2D)	(None, 7, 7, 4096)	102764544
fc2 (Conv2D)	(None, 7, 7, 4096)	16781312
conv2d_3 (Conv2D)	(None, 7, 7, 1000)	4097000
upsample_1 (Upsample)	(None, 224, 224, 1000)	0
=====		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

到目前为止，我们已经完成了 FCN 网络模型的定义和生成 FCN 网络模型对应的权重参数文件，但有一个细节没有讲到：通过对最后一层卷积后得到的热图（Heat Map）进行反卷积操作或者双线性插入法恢复原图大小，用于最后图像像素级的语义分割。

本例中没有采用反卷积操作，而是使用双线性插值放大热图。如【代码清单 6-6】所示，Upsample 层网络的输入特征图大小为 [7,7,1000]，输出大小为 [224,224,1000] 的特征作为 FCN 网络层的输出。该代码按照 Kears 的网络层定义模式编码，其核心部分是 `X = tf.image.resize_bilinear(X, new_shape)`，直接采用 TensorFlow 的双线性插值函数放大输入的特征 X 到 new\_shape 大小。



## 【代码清单 6-6】Upsample 层放大热图

```

from keras import backend as K
from keras.engine.topology import Layer
import tensorflow as tf

class Upsample(Layer):
    """
    Upsample 层
    """
    def __init__(self, size=(1,1), target_size=None, **kwargs):
        self.size = tuple(size) if size is not None else None
        super(Upsample, self).__init__(**kwargs)

    def call(self, x):
        x = self._resize_bilinear_with_factor(x, self.size)
        return x

    def _resize_bilinear_with_factor(self, x, factor_size=None):
        """
        通过双线性插值对 heat map 进行放大
        """
        # 计算放大后图片的尺寸
        height_factor, width_factor = factor_size
        height = x.shape[2] * height_factor      # 放大后图像的高
        width = x.shape[3] * width_factor         # 放大后图像的宽
        new_shape = tf.constant(np.array([height, width]).astype('int32')) # 放大后的图片

        # 使用 TensorFlow 中的双线性插值方法
        X = K.permute_dimensions(x, [0, 2, 3, 1])
        X = tf.image.resize_bilinear(X, new_shape)
        X = K.permute_dimensions(X, [0, 3, 1, 2])

        return X

# 测试
# X = Upsample(size=(32,32))(fcn_vgg16)

```

## 6.1.4 全卷积网络语义分割

准备好 FCN 网络模型和参数文件后，最后一步是对输入的图像进行语义分割。在【代码清单 6-7】中读入图像路径，然后使用 `predict()` 函数预测图片矩阵。在 `load_`

`image()` 函数读入图像路径后, 调用 Keras 的 `preprocess_input()` 函数对输入的图片矩阵进行预处理, 主要是用每个通道的像素值减去 ImageNet 数据集中每个通道的平均像素, 对输入图像矩阵进行归一化操作。

### 【代码清单 6-7】对输入的图片进行 FCN 图像语义分割

```
from PIL import Image
from keras.preprocessing import image
from keras.applications.imagenet_utils import preprocess_input

def load_image(img_path, show=False):
    """
    读取图片路径, 并把图片转换为 kears 可读矩阵
    """
    img = image.load_img(img_path, target_size=(512,512))
    img_tensor = image.img_to_array(img)

    # change shape(3,512,512) to shape(1,3,512,512)
    img_tensor = np.expand_dims(img_tensor, axis=0)
    img_tensor = preprocess_input(img_tensor)

    if show:
        show_img = np.transpose(img_tensor[0], (1,2,0))
        plt.imshow(show_img)
        plt.show()

    return img_tensor

# 读取 ImageNet 中的图片
image = load_image("image/2007_000129.jpg")

# 使用 fcn32s 预测图片
pred = fcn32_vgg16.predict(image, batch_size=1)
print("predict finished.")
```

输入图像大小为 `[512,512,3]`, `fcn32_vgg16()` 函数预测的输出特征矩阵大小为 `[512,512,1000]`。其中, 1000 代表矩阵的深度 (即 1000 个分类), 【代码清单 6-8】通过 `np.argmax()` 操作获得每一个像素在 1000 个分类概率中的最大值, 得到大小为 `[512,512,1]` 的图像, 如图 6-6 (c) 所示。

### 【代码清单 6-8】输出 FCN 语义分割效果

```
>>> image_pred = np.argmax(np.squeeze(pred), axis=0).astype(np.uint8)
>>> plt.imshow(image_pred)
>>> plt.show()
```

## 206 | 第6章 卷积神经网络进阶示例

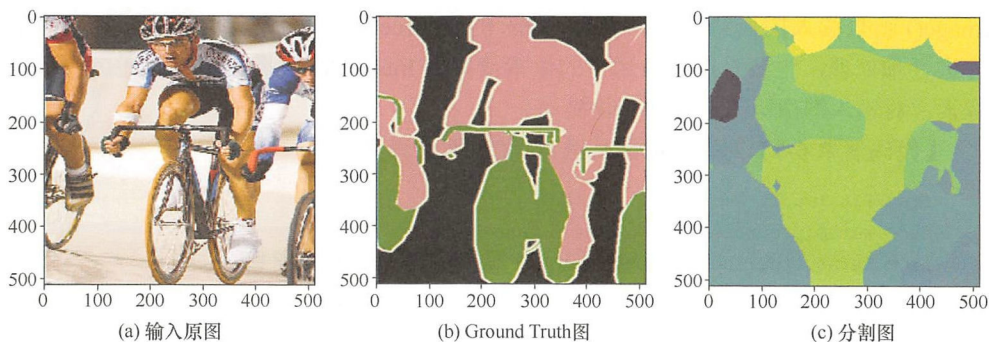


图 6-6 使用 ImageNet 数据集训练的 VGG16 转换 FCN 后得到的分割图，其中 ImageNet 分为 1000 个类别

从图 6-6 中可以看出，输入的原图为运动员在骑自行车，经过 FCN-32s 的预测分割后，得到图 6-6 (c) 中只能看出中间有一物体，其他边缘的地方均没有正确分割出来，该效果不尽人意。

实际上，因为原来 VGG16 的网络模型针对 1000 个类别进行分类，分类类别越多，过度拟合的情况就越高，FCN 的分割效果就会随之下降，因此 FCN-32s 网络模型的效果较差。接下来尝试重新定义 FCN 的网络模型，把最终的输出分类改为 Pascal 数据集中的 21 类，这次的输出不直接进行上采样操作为输入原图大小，而是进行 2 倍的上采样操作，方便传给 FCN-16s 进行操作。

#### 【代码清单 6-9】FCN-32s 模型定义

```
def FCN32s_Vgg(input_shape=None):
    ...
    ...

    # Convolutional layers transfered from fully-connected layers
    x = Conv2D(4096, (7, 7), activation='relu', padding='same', name='fc6')(x)
    x = Dropout(0.5)(x)
    x = Conv2D(4096, (1, 1), activation='relu', padding='same', name='fc7')(x)
    x = Dropout(0.5)(x)
    x = Conv2D(21, (1, 1), activation='linear', padding='same', name='score_fr')(x)

    # 线性插值进行上采样 2 倍操作
    X = Upsample(size=(2, 2))(x)

    model = Model(img_input, X)
    return model
```

#### 【代码清单 6-10】FCN-32s 模型输出

```
>>> fcn32s = FCN32s_Vgg(input_shape=(3, 512, 512))
```

```
>>> print('Model loaded.')
>>> fcn32s.summary()
```

Layer (type)	Output Shape	Param #
=====		
input_5 (InputLayer)	(None, 224, 224, 3)	0
<hr/>		
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
<hr/>		
.....		
.....		
.....		
<hr/>		
fc6 (Conv2D)	(None, 16, 16, 4096)	102764544
<hr/>		
dropout_5 (Dropout)	(None, 16, 16, 4096)	0
<hr/>		
fc7 (Conv2D)	(None, 16, 16, 4096)	16781312
<hr/>		
dropout_6 (Dropout)	(None, 16, 16, 4096)	0
<hr/>		
score_fr (Conv2D)	(None, 16, 16, 21)	86037
<hr/>		
score2 (Conv2DTranspose)	(None, 32, 32, 21)	7077
=====		
Total params: 134,353,658		
Trainable params: 134,353,658		
Non-trainable params: 0		

从 FCN-32s 网络模型到 FCN-16s 网络模型，该问题会变得很简单。如【代码清单 6-11】所示，`fcn_32to16()` 函数读入 FCN-32s 模型，提取最后卷积层的特征和 Block4 的 Pooling 层得到的特征，把两层特征向量求和，得到大小为 [32,32,21] 的求和特征。最后把该特征进行 16 倍的上采样操作，最终作为 FCN-16s 的输出，输出结果见【代码清单 6-12】。

【代码清单 6-11】把 FCN-32s 网络结构改为 FCN-16s 网络结构

```
def fcn_32to16(fcn32model=None):
    # 边界检查
    if fcn32model is None:
        raise Exception("the mode should not be null.")
```



```
# score Pooling 层
sp4_ = Conv2D(21, (1, 1), padding="same", activation=None, name='score_pool4')
sp4 = sp4_(fcn32model.layers[14].output)

# score Pooling 层
sp5 = fcn32model.layers[-1].output

# 合并 pool4 和 pool5 层的特征
sum_sp = merge([sp4, sp5], mode='sum')

# 线性插值进行上采样 16 倍操作
X = Upsample(size=(16, 16))(sum_sp)

model = Model(fcn32model.input, X)
return model
```

【代码清单 6-12】输出网络模型的定义

```
>>> fcn_vgg16 = fcn_32to16(fcn_vgg32)
>>> fcn_vgg16.summary()
```

Layer (type)	Output Shape	Param #	Connected to
=====			
input_3 (InputLayer)	(None, 512, 512, 3)	0	
conv1_1 (Conv2D)	(None, 512, 512, 64)	1792	input_3[0][0]
conv1_2 (Conv2D)	(None, 512, 512, 64)	36928	conv1_1[0][0]
block1_pool (MaxPooling2D)	(None, 256, 256, 64)	0	conv1_2[0][0]
.....			
.....			
.....			
fc6 (Conv2D)	(None, 16, 16, 4096)	102764544	block5_pool[0][0]
dropout_5 (Dropout)	(None, 16, 16, 4096)	0	fc6[0][0]
fc7 (Conv2D)	(None, 16, 16, 4096)	16781312	dropout_5[0][0]
dropout_6 (Dropout)	(None, 16, 16, 4096)	0	fc7[0][0]

```
score_fr (Conv2D)          (None, 16, 16, 21)   86037      dropout_6[0][0]
score_pool4 (Conv2D)       (None, 32, 32, 21)   10773      block4_pool[0][0]
score2 (Conv2DTranspose)   (None, 32, 32, 21)   7077       score_fr[0][0]
merge_3 (Merge)           (None, 32, 32, 21)    0          score_pool4[0][0] score2[0][0]
upsample_new (Conv2DTranspose) (None, 512, 512, 21) 451605     merge_3[0][0]
=====
Total params: 134,816,036
Trainable params: 134,816,036
Non-trainable params: 0
```

图 6-7 所示为 FCN-16s 在 21 个分类样本中的语义分割效果。该分割效果较为理想，图（b）和（d）分别为图（a）和（c）的分割结果，从图中可以清晰地看出分割为运动员和单车两个类别。

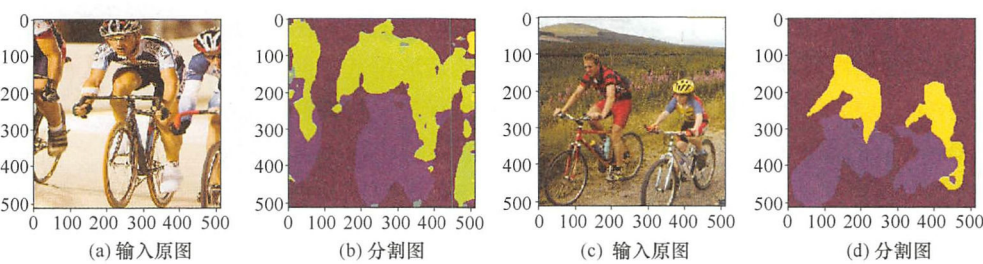


图 6-7 使用 Pascal 数据集训练的 VGG16 转换 FCN 后得到的分割图，其中 Pascal 数据集分为 21 个类别

## 6.2 示例2: 深度可视化网络

神经网络在近年来成了深度学习的代名词，而深度学习成了人工智能的代名词，因为深层神经网络在最新的学术研究上已经取得了惊人的成就！可随之带来的疑问也越来越多，深度神经网络为什么会有这么高的分类预测准确率？卷积神经网络对输入图像提取到的是什么样的纹理？其网络中每一层到对图像进行了什么改变？网络中不同层的卷积核有什么区别？每一层卷积得到的特征图又是什么样的？它真的是一个黑盒子吗？

长久以来，卷积神经网络被称为是黑盒子（Black Boxes），这意味着其内部工作

原理是神秘莫测的。近年来，(Erhan et al. 2009) 和 (Jason Yosinski et al. 2015) 等研究学者对卷积神经网络的权重文件进行研究分析，以寻求更好的方式去解析每个神经元所代表的内容，以及每个卷积核学到的高维特征。

卷积神经网络虽然在一定程度上可以视为黑盒子，但我们总是希望能够知道更多关于它的内容，深入探索其内部工作原理。本例将会以 VGG16 网络模型作为基础，通过梯度上升算法，可视化每一个卷积核希望看到的内容，以帮助我们进一步了解卷积神经网络的内部网络模型。

## 6.2.1 梯度上升法

根据 (Jason Yosinski et al. 2015) 的定义：有一张图  $A$ ，能够使其对应某卷积层的卷积核具有最高的激活值，那么图  $A$  是该卷积核“想要看到的”或者“正在寻找的纹理特征”。我们希望找到图像  $A$  经过卷积神经网络传播到指定卷积核时，该图可以使得该卷积核的激活值最高，那么这张图就是该指定卷积核希望得到的纹理特征。

为了得到上述的图  $A$ ，我们使用如图 6-8 所示的噪声图  $x$  作为卷积神经网络的输入进行向前传播，然后取得其在网络中第  $i$  层  $j$  个卷积核的激活值  $a_{ij}(x)$ ，接着进行反向传播计算激活值的梯度  $\partial a_{ij}(x) / \partial x$ ，最后用该指定的卷积核梯度值来更新噪声图  $x$ 。通过不断迭代更新噪声图  $x$ ，改变图  $x$  每个像素的颜色值，以增加对该卷积核的激活，最终使得图  $x^*$  对于在卷积神经网络中第  $i$  层  $j$  个卷积核具有较高的激活值。上述过程就是使用梯度上升算法，还原卷积核想要提取的高维特征：



图 6-8 随机产生的噪声图

$$x \leftarrow x + \eta \frac{\partial a_{ij}(x)}{\partial x} \quad (6-1)$$

其中  $\eta \in \mathbf{R}$  为梯度上升的学习率。

下面使用代码生成噪声图作为网络的输入，来看通过式 (6-1) 的梯度上升算法最后得到的图像。在【代码清单 6-13】中，通过 Keras 的后端函数获取指定层 `layer_name` 中的损失值，该损失值将用于最大化某个指定卷积核的激活值。其中，函数 `gradients(loss, variables)` 为返回 `loss` 关于 `variables` 的梯度。

**【代码清单 6-13】** 获取目标卷积核的梯度

```
from keras import backend as K

layer_name = 'block5_conv1'
```

```

# filter_index 可以从 0-511 的整数, 因为 'block5_conv1' 层有 512 个卷积核
filter_index = 0

# 建立一个损失函数, 该损失函数能够最大化该层中第 index 个卷积核的激活值
layer_output = layer_dict[layer_name].output
loss = K.mean(layer_output[:, filter_index, :, :])

# 通过损失值计算输入图像的梯度
grads = K.gradients(loss, input_img)[0]

# L2 正则化对梯度进行 L2 正则化
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)

# 使用了 keras 后端函数, 用于输入图像矩阵返回对应的梯度和损失
iterate_fun = K.function([input_img], [loss, grads])

```

上述代码加入了一个小技巧——L2 正则化操作, 使权重参数的更新值不会过大, 好处是使得梯度上升的过程能够平滑进行。

后端函数 `function` 用传递来的参数实例化一个 Keras 的 `function` 类返回。这相当于将 `function` 的对象当作函数来使用, 重载了括号运算符 (如 `outputs = self.train_function(inputs)`)。根据刚刚定义的损失函数 `iterate_fun`, 现在可以根据式 (6-1) 对卷积核的激活值进行梯度上升计算, 如【代码清单 6-14】所示。

**【代码清单 6-14】产生一张噪声图并使用梯度上升操作**

```

import numpy as np

# 产生一个噪声图
input_img = np.random.random((1, 3, img_width, img_height))
input_img = (input_img - 0.5) * 20 + 128

mFSize = 5
mFEvery = 10
mIter = 201

# 开始迭代式进行梯度上升计算
for i in range(mIter):
    loss_value, grads_value = iterate([input_img_data])
    input_img += grads_value * step

    input_img = np.clip(input_img_data, 0., 255.)

    if mFSize is not 0 and i % mFEvery == 0 :

```



```

input_img_data = median_filter(input_img, size=(1, 1, mFSize, mFSize))

if i % 50 == 0:
    print('\t%d, Current loss value:%f' % (i, loss_value))

```

在进行梯度上升的过程中，我们对输出的矩阵进行了 Clip 操作和中值滤波操作。其中，Clip 操作为限制了图形矩阵值为 0 ~ 255，目的是减少输出的图像饱和度；中值滤波是为了让输出的图像更直观，对相邻像素进行平滑操作。【代码清单 6-15】实现上述操作，并进行可视化，最终得到如图 6-9 所示的图像。

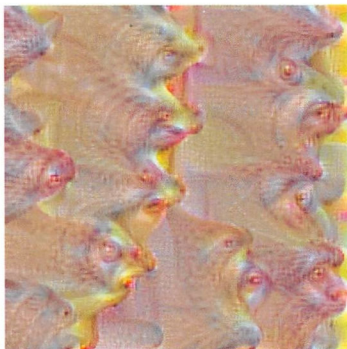


图 6-9 通过梯度上升法还原卷积核想要提取的高维特征

#### 【代码清单 6-15】显示该层网络中第 filter\_index 个可视化卷积核

```

from scipy.misc import imsave

# util function to convert a tensor into a valid image
def deprocess_image(x):
    # normalize tensor: center on 0., ensure std is 0.1
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    # clip to [0, 1]
    x += 0.5
    x = np.clip(x, 0, 1)

    # convert to RGB array
    x *= 255
    x = x.transpose((1, 2, 0))
    x = np.clip(x, 0, 255).astype('uint8')
    return x

```

```
# decode the resulting input image
img = deprocess_image(input_img_data[0])
kept_filters.append((img, loss_value))

plt.imshow(kept_filters[0][0])
imsave('%s_filter_%d.png' % (layer_name, filter_index), img)
```

经过梯度上升算法对噪声图进行像素级别的计算后得到图 6-9，可以看出，该图能够有效地分辨出许多鱼头或者山羊头组成的特征图。同时，该图像经过 CNN 网络模型计算后，其所在分类索引号中能够获得最高的激活值，也就是图 6-9 所示的分类索引号所在的卷积核希望得到的纹理特征。例如，该图对应 VGG16 最后一层索引号为 100，经过最后一层第 100 个卷积计算后，输出的激活值将是最高的，这代表最后一层第 100 个卷积核希望得到输入图纹理特征。

值得一提的是，这里对卷积核进行可视化并不是指可视化指定卷积核本身，而是通过分析网络参数文件和梯度上升算法，还原指定卷积核想要提取到的高维特征。

## 6.2.2 可视化所有卷积层

可视化所有卷积层想要得到的高维特征是本例中最有趣的部分。通过分析卷积神经网络模型中的卷积核，将会看到每一层卷积核提取的内容、纹理、特征。当我们深入了解卷积神经网络模型提取特征背后的意义，就可以有足够信心和知识去修改卷积神经网络的参数。

下面利用已经训练好的 VGG16 网络模型，来系统地可视化各个网络层的各个卷积核，看看卷积神经网络对输入进行逐层分解，希望提取到的特征。

在可视化某层的所有卷积核前，需要在【代码清单 6-16】中指定可视化的卷积层和该层的卷积核数。

### 【代码清单 6-16】可视化所有卷积核前的工作

```
# 对于 vgg16 模型获取每一层的名字和信息存储在 dict 中
layer_dict = dict([(layer.name, layer) for layer in model.layers[1:]])

layer_name = 'block1_conv1'

# 输入层信息
input_img = model.input

# 输出层信息
layer_output = layer_dict[layer_name].output
```

```
# 从 512 个卷积核中随机地挑选出 64 个作为可视化对象
all_filter = [x for x in range(512)]
filter_sample = random.sample(all_filter, 64)
```

最后代码的执行结果如图 6-10 所示，Block1\_Conv1 的卷积核主要完成如颜色、方向等组成，到了 Block2\_Conv2 的卷积核明显比 Block1\_Conv1 会有更多的纹理和不同的纹理方向，所表达的颜色也更加丰富多样，并且可以在边缘处看到有部分凹凸表现。

Layer Block1\_Conv1



Layer Block2\_Conv1

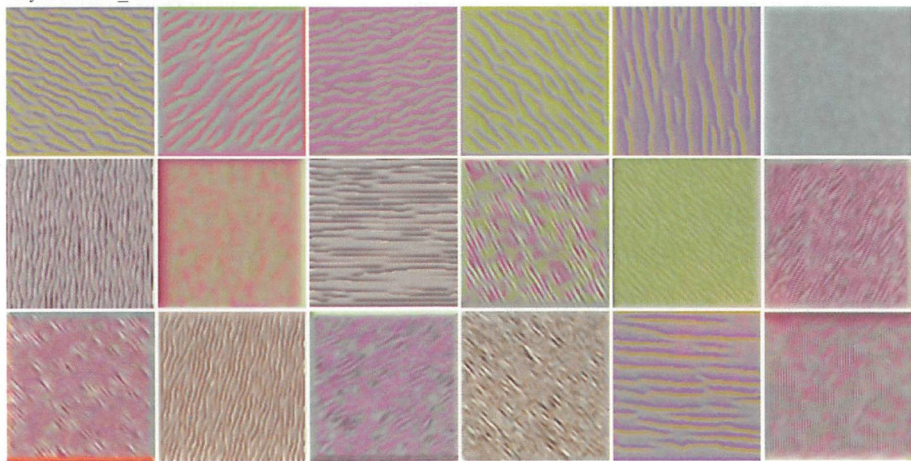


图 6-10 VGG16 Block1\_Conv1、Block2\_Conv1 部分卷积核可视化

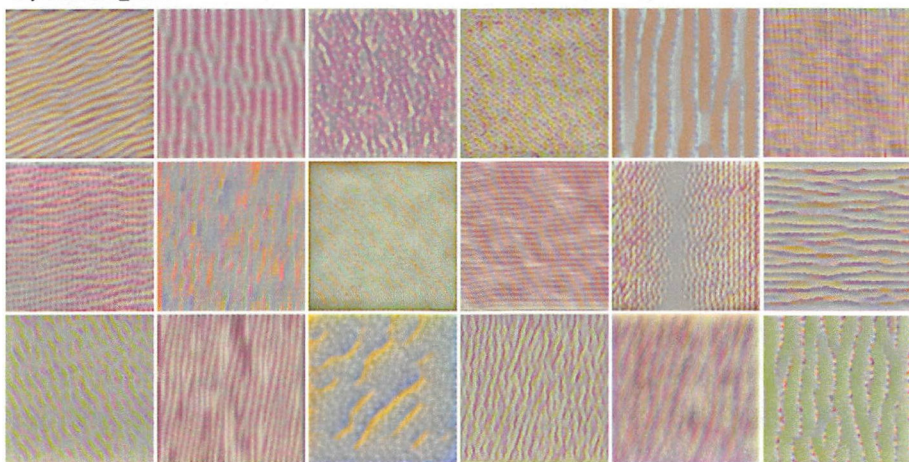
随着对 VGG16 网络模型的继续深入，这些颜色和方向与基本的纹理进行组合，逐渐生成特殊纹理。当进入 Block3\_Conv1 后，方向和颜色的表现开始变少，并



开始出现更加复杂的纹理特征（圆形、螺旋形、多边形、波浪等形状组合），到了Block5\_Conv1后可以清晰地看到其纹理更加特别，随着网络空间信息的增长而出现了更加精细和复杂的特征。

如图6-11所示，图像变得越来越复杂，因为卷积操作开始纳入越来越大的空间范围信息，所呈现和表达的信息进一步丰富。部分读者经过实际操作之后或许会发现：在同一卷积层中会出现少量的可视化图是空白或者相同的情况，这意味着该层卷积核对后续的操作并没有产生实际的作用，可以去掉网络中部分卷积核，以减少神经网络的计算量和过度拟合的可能性。

Layer Block3\_Conv1



Layer Block4\_Conv1

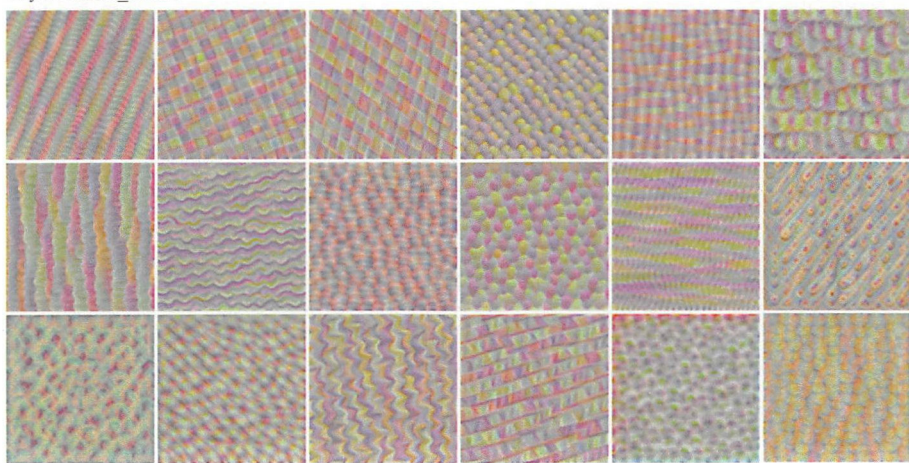


图 6-11 VGG16 Block3\_Conv1、Block4\_Conv1、Block5\_Conv1 部分卷积核可视化



Layer Block5\_Conv1

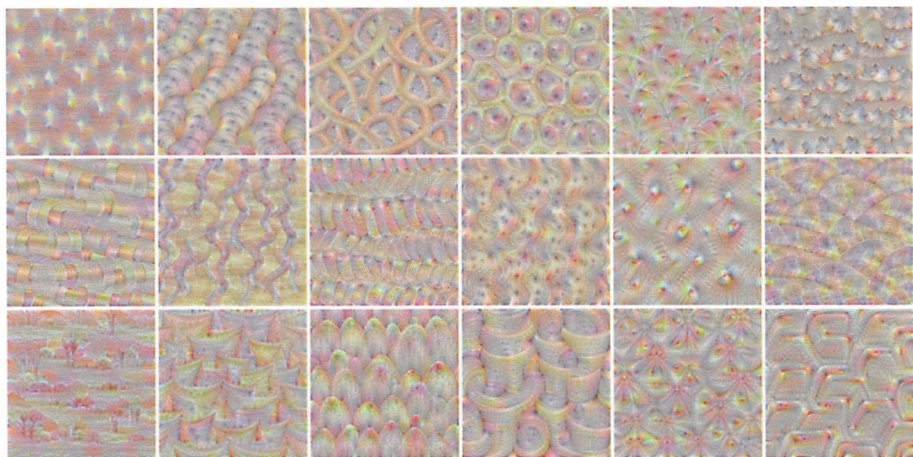


图 6-11 VGG16 Block3\_Conv1、Block4\_Conv1、Block5\_Conv1 部分卷积核可视化（续）

另外，也会有部分可视化图可以通过旋转平移获得另外一个可视化图。这是一个很有趣的研究方向，我们或许可以通过寻找一种旋转不变性的方法来潜在地代替网络层中的其他卷积核，从而压缩卷积核的数量。令人惊讶的是，即使对于级别相对高的滤波器，如 Block4\_Conv1 中，通过旋转、平移也可获得相同的可视化图。

如图 6-12 所示，在最高层（Block5\_Conv2、Block5\_Conv3）网络中，可以开始看到类似于分类对象中的某些纹理，如羽毛、眼睛、树叶等，纹理信息对比上几层更加丰富了。

Layer Block5\_Conv2

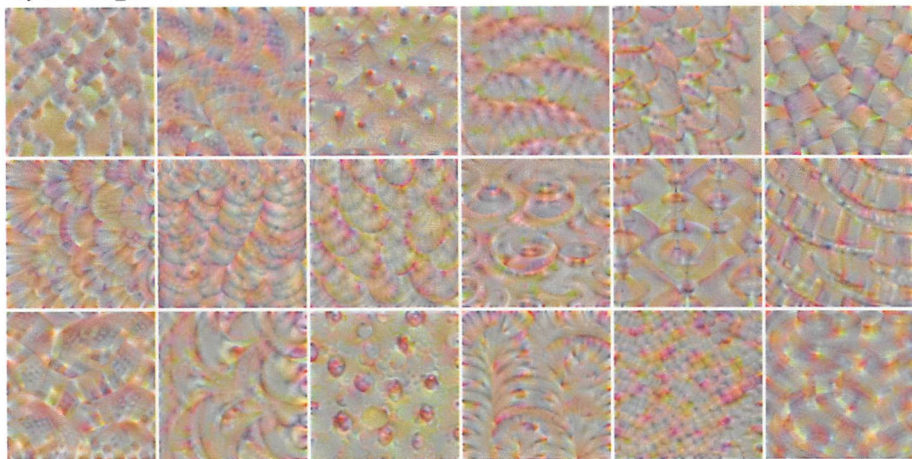


图 6-12 VGG16 Block5\_Conv2、Block5\_Conv3 部分卷积核可视化

Layer Block5\_Conv3

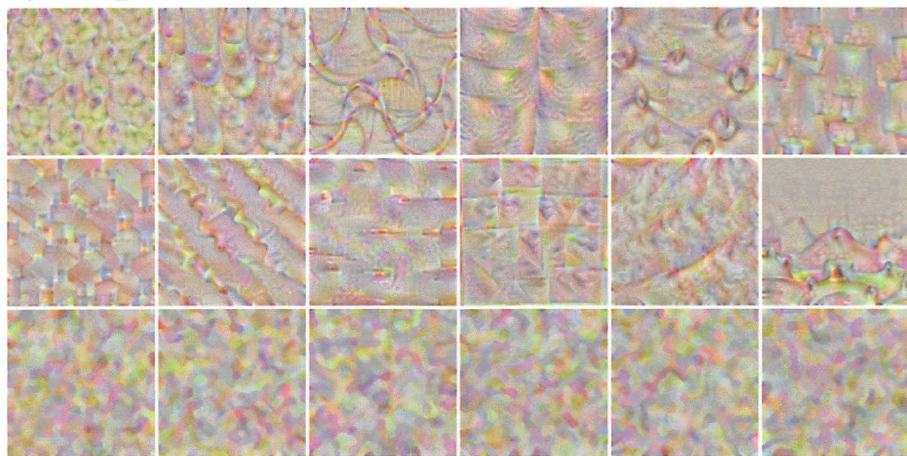


图 6-12 VGG16 Block5\_Conv2、Block5\_Conv3 部分卷积核可视化（续）

经过代码实践之后，我们发现在 Block5\_Conv3 层的 512 个卷积核里面只有 65 个卷积核的激活值是有效的，也就是其余的卷积核已经不能再继续提取高维纹理特征信息了，这是为何？（如图 6-12 所示，Block5\_Conv3 最后一行可视化卷积核图像只是由噪声图经过多次滤波操作得到的。）

VGG 网络并不能保证每一个卷积核都参与网络的计算，提取到高维特征。另外，当神经元饱和之后，网络把输入的特征直接输出，没有起到提取特征的作用。于是 ResNet 或者 GoogleNet 的作用就体现了，图 6-13 所示为 ResNet 的 Residual 的结构，VGG 的 Block5\_Conv3、Block5\_Conv2 开始出现大量没有用的卷积核，当模型的层次加深时，错误率却提高了，而 Block4\_Conv3 却有很多有用的信息可以向后传递。在这里读者也可以自己尝试着实现 ResNet 网络，然后通过梯度上升进行可视化分析，研究不同网络提取到的高维特征之间的区别，有助于进一步理解卷积神经网络 CNN 内部的工作方式。

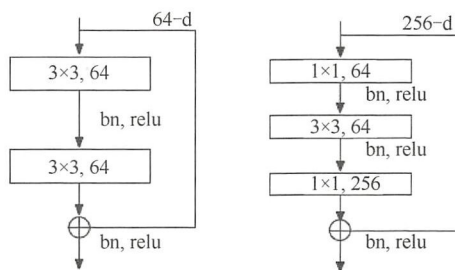


图 6-13 ResNet 的示例层

## 6.2.3 可视化输出层

在对输出分类层进行可视化操作之前，我们需要对 VGG16 进行一些特殊的操作：把 VGG16 网络的最后三层全连接层换成卷积层，也就是把卷积神经网络变成 FCN 网络，如【代码清单 6-17】所示。

【代码清单 6-17】可视化输出层

```
# 对 VGG16 最后的全连接层修改为卷积层
x = Conv2D(4096, (7, 7), activation='relu', padding='same', name='fc1')(x)
x = Conv2D(4096, (1, 1), activation='relu', padding='same', name='fc2')(x)
x = Conv2D(1000, (1, 1), activation='linear', name='predictions_1000')(x)
#x = Reshape((7,7))(x)

# 创建模型
model = Model(img_input, x)
weights_path = "fcn_vgg16_weights_tf_dim_ordering_tf_kernels.h5"

# 对应修改后的卷积层的权重参数进行重新排列
if os.path.isfile(weights_path) == False:
    flattened_layers = model.layers
    index = {}
    for layer in flattened_layers:
        if layer.name:
            index[layer.name]=layer

    for layer in vgg16.layers:
        weights = layer.get_weights()
        if layer.name=='fc1':
            # weights[0] = np.reshape(weights[0], (7,7,512,4096))
            weights[0] = np.reshape(weights[0], (7,7,512,4096))
        elif layer.name=='fc2':
            weights[0] = np.reshape(weights[0], (1,1,4096,4096))
        elif layer.name=='predictions':
            layer.name='predictions_1000'
            weights[0] = np.reshape(weights[0], (1,1,4096,1000))
        if layer.name in index:
            index[layer.name].set_weights(weights)
    model.save_weights(weights_path)

return model
```

最后经过梯度上升后可可视化卷积核图像如图 6-14 所示，使用到 ImageNet 分类类别中的 1、8、18、388、351、327、413、672，对应图中的每一格。从图 6-14 第一



张可视化图中可以看到，这张图中有一些如分类 goldfish（金鱼）的形状，又好像有很多条金鱼尾巴，图像中间还相夹着许多鳞片。虽然看上去这些图片很奇怪，但是假如把这些经过梯度上升后的可视化图片作为该 VGG16 网络模型的输入，其输出为对应类别的概率将会是最高的。

经过实际代码测试，经过 100 次迭代预测输入图 6-14 左上角的 goldfish 图给 VGG16 模型，每次预测的分类均为 1，100 次预测概率均高于 95%。

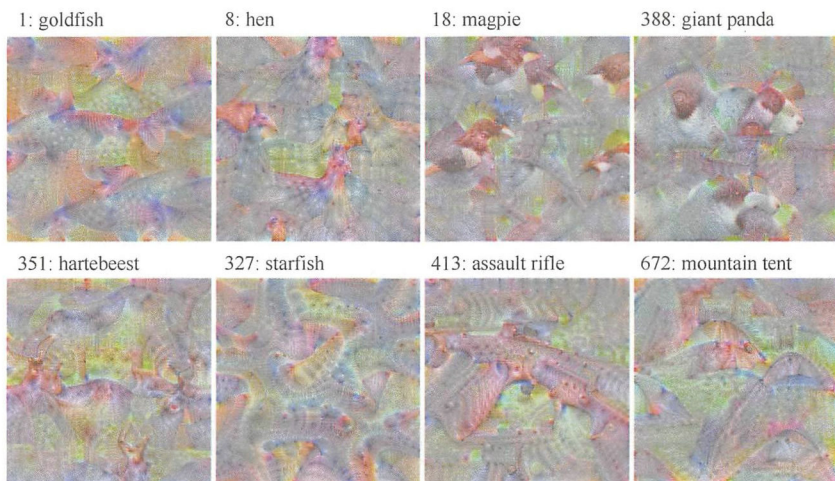


图 6-14 可视化输出层的卷积核

## 6.2.4 卷积神经网络真能理解视觉吗

卷积神经网络的两个主要作用为：

- 把输入的视觉空间图像，解耦成分层次的卷积核组合；
- 通过分层次的卷积核把输入的数据映射到不同的几何空间。

有人会宣称卷积神经网络通过分层思想对输入的图像进行解耦，这一过程模拟了人类视觉皮层，因此才能获得如此精妙的效果。可事实并不是这样，从科学的角度来看，这并不意味着我们真能在某种程度上解决计算机视觉的问题，而只是使用了数学的方法对输入的图像数据进行抽样和几何空间映射。即使科学是这样解释，但是科学家们也没有证据去证明视觉皮层不是这种工作方式。

深度学习虽然不能表达真正意义上的智力，但毋庸置疑的是其预测效果十分惊人，以至于在近年来没有其他算法可以比拟，甚至在某些情况下超过了人类的预测精度！我们不应期待着算法学习人类的思考方式，而应去拥抱数学，用其特殊的方法





式去为人类服务，继续发现、继续创造、继续在模拟数字领域领跑！

### 6.3 示例3：卷积神经网络艺术绘画

在人工智能领域里，在游戏中胜出和创造艺术一直被人们认为是难以用电脑完成的任务。但自从有了深度学习之后，这两项艰巨的任务终于得以被计算机所实现。想象一下，以后漫画画家完成手稿后不需要人工填充颜色，使用卷积神经网络模型就可以自动上色，然后再加以少量人工修改，这将会节省画家宝贵的时间；以后人工CG或许还能够用卷积神经网络模型去模拟、三维物体的建模由卷积神经网络独立完成，等等。下面就让我们一起去深入探索卷积神经网络进行艺术绘画的奥秘。

本例是卷积神经网络中最具有挑战性的例子，也是深度学习里面一个具有里程碑意义的例子——使用卷积神经网络进行艺术绘画。这里所指的卷积神经网络进行艺术绘画不是让网络模型凭空去创造艺术，而是临摹艺术，模仿画家的画风进行绘画（如图6-15所示）。

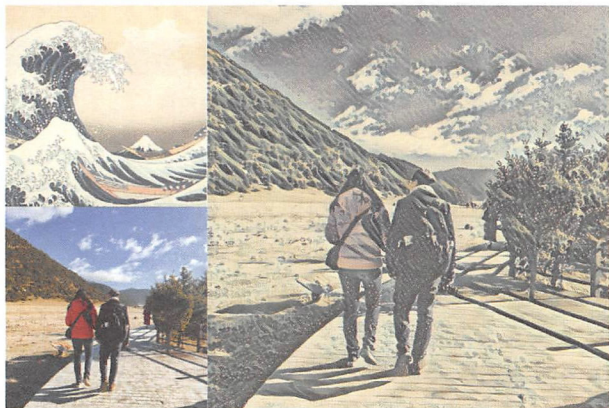


图 6-15 使用卷积神经网络进行风格转移。其中左上角为著名日本画家天明屋尚（Tenmyouya Hisashi）的名画《浪》，左下角为需要风格转移的原始图像，右侧为最终经过风格转移后产生的图像

使用卷积神经网络进行艺术绘画这一创新性的工作由 Leon A. Gatys、Alexander S. Ecker、Matthias Bethge 发表在 2016 年的 CVRP 会议上。后来 Leon A. Gatys 等人还成立 DeepArt 公司，并与 Adobe 公司进行深度合作，为 Adobe 提供数字化画风技术。同期 App Store 上推出一个名为 Prisma 的应用程序，输入图像后能够将画风转移，如图 6-16 所示。有兴趣的读者可以下载该应用程序去尝试不同风格的转换图，感受



卷积神经网络进行艺术绘画的精彩!



图 6-16 Prisma 软件风格转移效果图。其中每一副小图的左上角均为待学习的原始风格图像，大图为最终经过风格转移后产生的图像

### 6.3.1 算法思想

卷积神经网络模仿画风，实际上就是对图像进行风格转换。其目的是让最后合成图像  $X$  有输入原图内容并且带有画家风格。例如，给出一张内容图  $P$  和一张风格图  $A$ ，目标是找到一张合成图  $X$ ，让图  $X$  与图  $P$  有着相似的内容，同时又与图  $A$  有着相似的风格。

实际上图像风格转换也可以表示成数学上的优化问题。我们需要找一个方法去衡量两张图像内容上的不同，这就意味着目标是要找到一个函数。该函数能够让两张输入图像  $(X, P)$  的内容差异尽可能的小，并且这个函数值随着两张输入图像内容偏差增大而增加，该函数被称为内容损失函数：

$$\mathcal{L}_{\text{content}}(\text{Image 1}, \text{Image 2}) \approx 0 \quad (6-2)$$

同样我们需要找到一个方法去衡量两张图像风格上的不同，即找到一个函数来检测两张图像  $(X, A)$  风格的相似度。两个图像的风格越相似，其函数值越小，否则函数值增大。该函数被称为风格损失函数：

$$\mathcal{L}_{\text{style}}(\text{Image 1}, \text{Image 2}) \approx 0 \quad (6-3)$$

假设现在已经找到上述两个损失函数，这时图像风格转移问题就明确了。我们的目标就是找到一张图  $X$  在内容上尽可能与内容图  $P$  相似，风格上尽可能与风格图  $A$  相似。使用数学表达就是希望找到一个使内容损失函数式 (6-2) 和风格损失函数式 (6-3) 都尽可能小的  $X$ ，于是有了风格转移损失函数：

$$x^* = \arg \min_x (\alpha \mathcal{L}_{\text{content}}(X, P) + \beta \mathcal{L}_{\text{style}}(X, A)) \quad (6-4)$$



其中,  $\alpha, \beta \in \mathbf{R}$  用来控制合成的图像  $X$  与风格和内容损失值的权重大小。剩下的工作就是利用卷积神经网络迭代优化输出。

### 6.3.2 图像风格定义

使用卷积神经网络去创造艺术最重要的一点就是模仿画家的风格。在这里, 画家的风格可以定义为图像的纹理, 例如图像中的局部结构和颜色。最后就把问题转变为如何通过卷积神经网络找到风格图  $A$  中的纹理属性。

卷积神经网络适合将图像作为输入, 输出可以是一个分类的结果, 也可以是一张或者多张图像。一张图像传入卷积神经网络某层后输出为特征图, 这些特征图往往保留着输入图像中较有代表性的特征, 例如边缘、角点、结构等信息。Leon A. Gatys 等在论文中提到了如何使用卷积神经网络提取人工纹理。如图 6-17 所示, 上面一行图片是原始图像, 下面一行图片是卷积神经网络提取的人工纹理图, 上下行虽然看上去差别不大, 但仔细观察会发现人工纹理图是无序的, 也就是人工纹理图丢失了空间位置信息, 保留了图形中的纹理信息。



图 6-17 人工纹理提取算法效果展示图。上面一行的 3 张图片均为原始纹理图像, 下面一行的 3 张图片均为经过人工纹理提取算法后得到的纹理图。对比上下两行图片, 可以清晰地看出原始纹理图片中的物体纹理为有序的, 经过人工纹理提取算法后的生成纹理图会产生一定程度的纹理顺序错乱

下面来看如何使用卷积神经网络求得图像的人工纹理。根据 (Gatys et al., 2015) 介绍, 图像纹理提取的输入是一张随机的噪声图  $X$ , 通过迭代更新该噪声图  $X$ , 其中只保留原图中的颜色和局部结构, 直至噪声图  $X$  与设定的规则误差越来越小。而这个设定的规则就是 Gram 矩阵。

【代码清单 6-18】利用 numpy 产生了一个随机的噪声图, 并且对噪声图的每一个像素减去 128, 进行归一化操作。





## 【代码清单 6-18】随机产生一张噪声图

```
x = np.random.uniform(0, 255, (1, w, h, 3)) - 128.
```

## Gram 矩阵

现在的目标是使用卷积神经网络提取图像的纹理，但特征图是原图经过卷积后得到的，所以特征图依然会保留原图中大部分的空间位置信息。为了剔除特征图的空间信息，只保留其特征，这里使用  $N \times N$  的 Gram 矩阵  $G$ ， $N$  是特征图在该层当中特征图的数量。例如 VGG 卷积层 5\_1 中有 512 个特征图，那么 Gram 矩阵  $G$  的大小为  $512 \times 512$ 。因此，迭代的工作就是让噪声图  $X$  尽可能与 Gram 矩阵  $G$  相似，最后迭代更新的工作过程可以演变为梯度下降的工作。不同的是，梯度下降算法更新的是噪声图  $X$  中的像素值，而不是整个网络中的权重参数。

下面来定义 CNN 中的 Gram 矩阵，在第  $l$  层中有  $N^l$  个不同的卷积核（即有  $N^l$  个不同的特征图），每个特征图是一个  $M^l$  的向量，因此所有的特征图可以存储为矩阵  $F^l \in \mathbb{R}^{N^l \times M^l}$ ，所以有  $F_{ik}^l$  是第  $l$  层中位置  $k$  的特征图  $i$ ，进一步有  $G_{ij}^l$  为第  $l$  层中特征图  $i$  和特征图  $j$  的内积：

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \quad (6-5)$$

假设  $\vec{a}$  和  $\vec{x}$  分别是原始风格图像和合成风格图像，那么  $G^{la}$  和  $G^{lx}$  是它们在第  $l$  层中对应的 Gram 矩阵，最终  $l$  层中的损失期望为：

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{i,j}^{la} - G_{i,j}^{lx})^2 \quad (6-6)$$

总损失函数为  $L_{\text{style}}(\vec{a}, \vec{x})$ ， $w_l$  是  $l$  层对应的权重向量：

$$L_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^L w_l E_l \quad (6-7)$$

【代码清单 6-19】为 Gram 矩阵的代码和风格损失的代码实现。

## 【代码清单 6-19】Gram 矩阵实现

```
def gram_matrix(x):
    """
    gram 矩阵 -- 特征图的外积矩阵，度量各个维度自己的特性以及各个维度之间的关系
    """
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style_features, comb_features):
    """
```





风格损失 (style loss) 的目的是在生成的图像中保持参考图像的风格  
其中对于风格的定义使用了风格和生成特征图的 gram 矩阵

```
'''
channels = 3
S = gram_matrix(style_features)
C = gram_matrix(comb_features)
size = w * h
return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

在纹理分割提取中，只剩下最后一个疑问：应该选择卷积神经网络中哪一层作为 Gram 矩阵  $G$ 。经过实际测试，在浅层网络中重构的纹理图不太具有代表性（如 VGG 中 conv1\_1），而在深层网络中能够较好地反映原图的纹理特征（如 VGG 的 pool4），因此最后选择结合多层特征作为 Gram 层。

原因有以下两点：

- 网络层数越深，提取到的特征越明显；
- 根据局部感知原理，网络层数越深，提取到的特征会越来越大。

在【代码清单 6-20】中选择 VGG 网络中每一个瓶颈层后的第一层卷积层的特征图作为 Gram 矩阵  $G^l$ ，并对每一层进行遍历，计算其 Gram 矩阵  $G^l$  和对应的损失，最后累加到总损失函数中。

【代码清单 6-20】风格损失所需要的特征层

```
feature_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1',
                  'block4_conv1', 'block5_conv1']

# 定义风格损失
# 遍历 feature_layers 卷积层，把其特征作为 " 内容损失函数 "
for layer_name in feature_layers:
    layer_features = vgg_dict[layer_name]
    style_features = layer_features[1, :, :, :]
    comb_features = layer_features[2, :, :, :]
    sl = style_loss(style_features, comb_features)
    loss += (style_weight / len(feature_layers)) * sl # 0.25
```

### 6.3.3 图像内容定义

现在距离使用卷积神经网络进行艺术绘画又近了一步，通过 Gram 矩阵可以求得图像中的风格，下面来看如何定义和求解图像的内容。其实图像内容的提取与图像风格的提取类似：提取图像内容不需要迭代式地求解，而是从网络中的特征图中通过计算提取出来的。



与图像风格定义相似, 在第  $l$  层中有  $N_l$  个不同的滤波器 (即有  $N_l$  个不同的特征图), 而每个特征图是一个  $M_l$  的向量, 因此所有的特征图可以存储为一个大的矩阵  $F^l \in \mathbf{R}^{N_l \times M_l}$ , 因此有  $F_{ij}^l$  是第  $l$  层中位置  $j$  的特征图  $i$ 。

$$L_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F_{i,j}^{lp} - F_{i,j}^{lx})^2 \quad (6-8)$$

#### 【代码清单 6-21】内容损失函数

```
def content_loss(base_features, comb_features):
    ''' 内容损失 (content loss) 目的是在生成的图像保留图像的内容 '''
    return K.sum(K.square(comb_features - base_features))
```

`content_loss()` 是内容损失函数, 这里使用了 VGGNet 网络中第 4 个瓶颈层的第二层卷积层的特征图作为内容, 去计算内容损失 (如【代码清单 6-22】所示)。

#### 【代码清单 6-22】内容特征

```
# 定义内容损失, 把 block4_conv2 层得到的特征作为 " 内容损失函数 "
loss = K.variable(0.) # 实例化损失函数
layer_features = vgg_dict['block4_conv2']
base_img_features = layer_features[0, :, :, :]
comb_features = layer_features[2, :, :, :]
loss = .5 * content_loss(base_img_features, comb_features)
```

### 6.3.4 算法实现

前面已经讲述了如何通过卷积神经网络的特征获得图像的风格和内容, 下面使用卷积神经网络计算损失函数最小值, 对合成图进行风格和内容两方面的约束进行修正。

如图 6-18 所示, 通过卷积神经网络进行风格转移的算法步骤可以归纳如下。

- (1) 风格图  $A$  通过 VGGNet 网络, 然后计算和保存 Gram 矩阵  $G$ 。
- (2) 内容图  $P$  通过 VGGNet 网络, 求得其内容特征图  $F$ 。
- (3) 随机产生一个噪声图  $X$ , 通过向后传播算法迭代更新噪声图  $X$ , 直到噪声图  $X$  满足  $G$  和  $F$  的约束为止。

约束式为总损失函数  $L_{\text{total}}(\vec{p}, \vec{x}, \vec{a})$ , 其中  $\alpha$ 、 $\beta$  分别为内容损失的权重和风格损失的权重:

$$L_{\text{total}}(\vec{p}, \vec{x}, \vec{a}) = \alpha L_{\text{content}}(\vec{p}, \vec{x}) + \beta L_{\text{style}}(\vec{a}, \vec{x}) \quad (6-9)$$

#### 1. 定义输入数据

首先输入的图像有 3 张, 第 1 张是内容图  $P$  (`base_img`), 第 2 张是风格图  $A$  (`style_`



img)，第3张是最后的合成图  $X$  (comb\_img)。使用 Kears 的 variable 将内容图  $P$  和风格图  $A$  实例化，并使用 Keras 的 placeholder 对空白合成图  $X$  实例化，然后把3张图合并成一个 Keras 的输入对象，如【代码清单 6-23】所示。

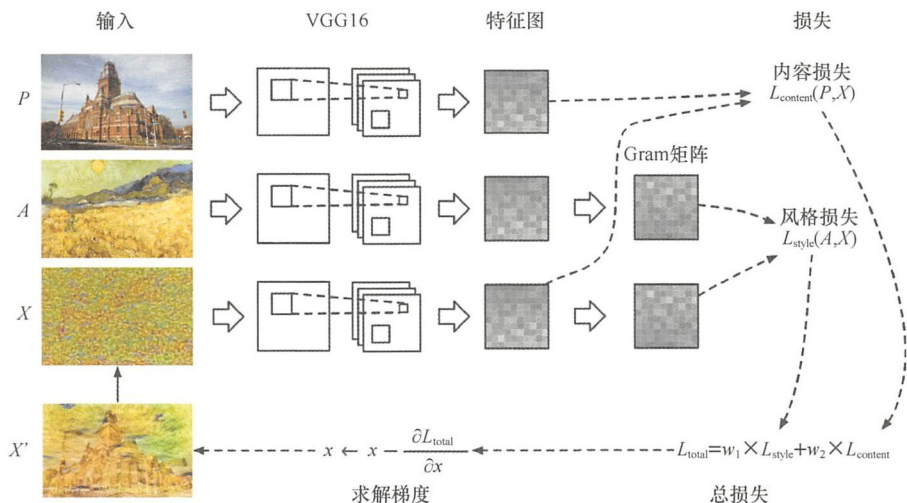


图 6-18 图像风格转移算法流程图。 $P$  为内容图像， $A$  为风格图像， $X$  为随机产生噪声图， $X'$  为正在迭代风格转移图。算法流程中对  $P$ 、 $A$ 、 $X$  都经过 VGG16 网络提取高维特征图，然后针对不同的目标设定不同的损失函数内容，通过自定义的损失函数使得梯度下降算法更新  $X'$  图像中的像素值，最终达到风格转移的目的

### 【代码清单 6-23】定义输入

```
# 定义输入的图像 P、A、X
base_img = K.variable(preprocess_image(base_img_path, imgx_size)) # 内容图 Content (P)
style_img = K.variable(preprocess_image(style_img_path, imgx_size)) # 风格图 Style (A)
comb_img = K.placeholder((1, img_nrows, img_ncols, 3)) # 目标合成图 (X)

# 把输入的三张图像合并成 keras 的 tensor 对象 -> shape=(3, h, w, 3)
input_tensor = K.concatenate([base_img, style_img, comb_img], axis=0)
```

## 2. 总损失函数

接下来，卷积神经网络的模型采用了 VGG16 网络模型，把  $P$ 、 $A$ 、 $X$  这 3 张图作为 VGGNet 网络模型的输入，并对 VGGNet 网络提取其每一层的信息，放入 vgg\_dict 字典里，作用是方便调用风格损失和内容损失函数，如【代码清单 6-24】所示。

### 【代码清单 6-24】VGGNet 网络字典

```
vgg_dict = dict([(layer.name, layer.output) for layer in model.layers])
```



现在回到一个更加简单的问题上，上面已经实现了风格损失和内容损失函数，剩下就是全局损失函数。为了让合成的图像保持图像的局部连贯性（Locally Coherent），也就是让图像更加平滑，需要在【代码清单 6-25】中加入 `total_variation_loss` 函数，合并风格损失和内容损失。

【代码清单 6-25】总损失函数定义

```
def total_variation_loss(x, size):
    assert K.ndim(x) == 4 # 输入要求为 4 维的数据否则报错

    row = size[0] - 1
    col = size[1] - 1
    a = K.square(x[:, :row, :col, :] - x[:, 1:, :col, :])
    b = K.square(x[:, :row, :col, :] - x[:, :row, 1:, :])
    return K.sum(K.pow(a + b, 1.25))

loss += 1.0 * total_variation_loss(comb_img, imgx_size) # 总损失函数的值 loss
```

### 3. 定义梯度和解决优化问题

现在引入一个 `Evaluator` 类，用于计算每一次的损失和梯度，通过两个独立的函数 `loss` 和 `grads` 来实现，如【代码清单 6-26】所示。单独计算损失和梯度的原因是 `scipy.optimize` 中的输入要求是单独输入损失和梯度，值得注意的是，单独计算损失和梯度效率并不高。

【代码清单 6-26】实例化损失和梯度的计算类

```
outputs = [loss]
outputs += grads
f_outputs = K.function([comb_img], outputs) # 实例化一个 Keras 函数
evaluator = Evaluator(imgx_size, f_outputs)

def eval_loss_and_grads(x, size, f_outputs):
    """ 计算损失和梯度 """
    x = x.reshape((1, size[0], size[1], 3))
    outs = f_outputs([x])
    loss_value = outs[0]
    grad_values = outs[1].flatten().astype('float64')

    return loss_value, grad_values

class Evaluator(object):
    """ Evaluator 类从两个不同的程序中分别获得 loss 损失和 gradients 梯度 """
    def __init__(self, size, f_outputs):
        self.loss_value = None
```





```

self.grads_values = None
self.size = size
self.f_outputs = f_outputs

def loss(self, x):
    assert self.loss_value is None
    loss_value, grad_values = eval_loss_and_grads(x, self.size, self.f_outputs)
    self.loss_value = loss_value
    self.grad_values = grad_values
    return self.loss_value

def grads(self, x):
    assert self.loss_value is not None
    grad_values = np.copy(self.grad_values)
    self.loss_value = None
    self.grad_values = None
    return grad_values

grads = K.gradients(loss, comb_img)

```

最后一步就是对图像进行迭代计算，这里设定迭代次数为 45。迭代的次数不是越多越好（因为会产生过度拟合、等待时间过长等问题），经过实际测试发现，图像风格转换的迭代次数控制在 10 次左右，能够产生比较理想的效果。在【代码清单 6-27】中，`cur_loss` 是当前的损失，`cur_grads` 是当前的梯度，代码中使用 `scipy` 里的 L-BFGS 算法（基于拟牛顿法 BFGS 优化算法的改进）计算损失函数的最小值。

### 【代码清单 6-27】优化损失函数

```

# 产生一张服从高斯分布的噪声图
x = np.random.uniform(0, 255, (1, height, width, 3)) - 128.
iterations = 45 # 迭代次数

for i in range(iterations):
    print('Start of iteration', i)

    cur_loss = evaluator.loss # 当前损失
    cur_grads = evaluator.grads # 当前梯度

    # 使用 scipy 的内置 L-BFGS 算法计算损失函数最小值
    x, min_val = scipy.optimize.fmin_l_bfgs_b(cur_loss, x.flatten(), fprime=cur_grads, maxfun=20)

```

如图 6-19 所示，第一次迭代的图只有风格图 **P** 中类似的颜色，没有图像内容和风格笔触。经过 10 次迭代之后，网络风格的转换效果越来越明显，到第 45 次迭代



都是在优化图像内容，为其补充更多图像内容细节。



图 6-19 使用卷积神经网络进行图像风格转换的迭代效果图。图中从左到右分别为经过 1 次、10 次、45 次迭代的不同的效果

如果读者对效果不满意，可以尝试去改变内容或者风格损失的权重，或者换一张图，也可以把风格和内容互换来看效果。

虽然上面使用卷积神经网络进行艺术绘画的效果已经颇为惊人，但是与应用程序 Prisma 相比还有很大的提升空间，我们还可以继续进一步处理细节，如超参数的设置、初始化问题等。值得一提的是，上面程序进行 10 次迭代需要等待一小段时间，而软件 Prisma 只需要数秒即可。为了解决图像风格转换需要消耗大量的计算资源这一问题，(Johnson et al, 2016) 提出了使用独立的卷积神经网络去模拟风格转换，最终使图像风格转换效率比原来提升了近 100 倍，有兴趣的读者可以进一步阅读文献来了解。

## 引用/参考

- [1] Gatys L A, Ecker A S, Bethge M. Texture Synthesis Using Convolutional Neural Networks[J]. 2015, 70(1):262-270.
- [2] Zeiler M D, Fergus R. Visualizing and Understanding Convolutional Networks[C]// European Conference on Computer Vision. Springer, Cham, 2014:818-833.
- [3] Yosinski J, Clune J, Nguyen A, et al. Understanding Neural Networks Through Deep Visualization[J]. Computer Science, 2015.
- [4] Mahendran A, Vedaldi A. Visualizing Deep Convolutional Neural Networks Using Natural Pre-images[J]. International Journal of Computer Vision, 2016, 120(4):1-23.
- [5] Drineas P, Mahoney M W. On the Nyström Method for Approximating a Gram Matrix for Improved Kernel-Based Learning[J]. Journal of Machine Learning Research, 2005, 6:2153-2175.
- [6] Zinkevich M. Online Convex Programming and Generalized Infinitesimal Gradient Ascent[J]. Icml, 2003:928-936.



- [7] Isola P, Zhu J Y, Zhou T, et al. Image-to-Image Translation with Conditional Adversarial Networks[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2017:5967-5976.
- [8] Gatys L A, Ecker A S, Bethge M. Image Style Transfer Using Convolutional Neural Networks[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE Computer Society, 2016:2414-2423.
- [9] Gatys L A, Ecker A S, Bethge M. Texture synthesis using convolutional neural networks[C]// International Conference on Neural Information Processing Systems. MIT Press, 2015:262-270.
- [10] Gatys L A, Ecker A S, Bethge M, et al. Controlling Perceptual Factors in Neural Style Transfer[C]// IEEE Conference on Computer Vision and Pattern Recognition. IEEE, 2017:3730-3738.
- [11] Funke C M, Gatys L A, Ecker A S, et al. Synthesising Dynamic Textures using Convolutional Neural Networks[J]. 2017.
- [12] Gatys L A, Ecker A S, Bethge M. Texture and art with deep neural networks.[J]. Current Opinion in Neurobiology, 2017, 46:178-186.



---

# 第 7 章

---

## 循环神经网络

本章主要内容：

- 循环神经网络概述
- 循环神经网络模型
- 时间反向传播算法
- 梯度消散与梯度爆炸问题





循环神经网络（Recurrent Neural Networks, RNN），因其循环递归处理历史数据和对历史记忆进行建模的特殊特性，适用于处理时间、空间序列上有强关联的信息。

普通神经网络和卷积神经网络在对损失函数的参数求导时，一般使用反向传播（BP）算法。在循环神经网络结构中求损失函数的参数梯度有很多中算法，其中时间反向传播算法（Back Propagation Through Time, BPTT）最为常用。随着 BPTT 在循环神经网络模型中计算，梯度传递过程会引起梯度消失（Gradient Vanish）或者梯度爆炸（Gradient Explosion）的问题。

随着循环神经网络模型的规模的增大，对序列数据的记忆能力就会逐渐下降。由于循环神经网络模型基本结构过于简单，不能存储长期记忆，当序列信号在网络中多次传递后，有可能引起梯度问题。因此基于循环神经网络模型，学者们又提出了长短期记忆（Long Short-Term Memory, LSTM）网络等更加复杂的循环神经网络和记忆单元，使得循环神经网络模型可以更加有效地处理更长的序列信号。

本章在介绍循环神经网络的基本网络模型的同时，会出现部分数学推导计算公式，如果读者可以投入些精力去阅读这些数学公式，对了解循环神经网络模型将会有非常大的帮助。另外，由于循环神经网络模型主要用途是处理序列数据，因此本章将会介绍如何通过深度学习框架去使用循环神经网络模型进行文本预测，和对股票收盘数据进行分析预测。接下来让我们一起去感受循环神经网络模型处理序列数据的魅力吧！

## 7.1 初识循环神经网络

循环神经网络是深度学习中的一个重要分支，近年来与循环神经网络模型相关的研究发展迅速。其中的成功案例包括手写字体识别、语音识别、自然语言处理和基于计算机视觉等序列问题。从生物神经学角度看循环神经网络，可以认为其是对生物神经系统环式链接（Recurrent Connection）的简单模拟，而这种环式链接在新大脑皮质（Neocortex）中是普遍存在的。这也从侧面反映人类学习是一个动态变化的过程，因而对该神经元的模拟在生物工程上有着重要的意义。

循环神经网络模型通常用于描述动态的序列数据，随着时间的变化而动态调整自身的网络状态，并不断循环传递，还可以接受广泛的序列信息结构作为输入。不同于前馈神经网络（例如 ANN、DNN、CNN 等），循环神经网络模型更加重视网络中的反馈作用。由于存在着当前状态与过去状态或者与未来状态的链接，循环神经网络模型可以具有一定的记忆功能。而到目前为止，具有代表性的递归神经网络不仅包括传统循环神经网络模型，还包括长短期记忆神经网络（Long Short-Term



Memory, LSTM) 以及门控循环单元 (Gated Recurrent Unit, GRU) 模型。

图 7-1 (a) 所示是普通前馈式深度神经网络模型, 图 7-1 (b) 所示是循环神经网络的一个模型。从图中可以看出, 普通的深度神经网络是从左到右逐层传递的, 其网络的神经元数据不断向前传递直到输出, 所在层 (当前层) 的神经元之间并没有连接关系; 而循环神经网络不同于前馈式的神经网络, 其引入了定向循环机制, 神经元之间互相依赖、互相连接, 因此能够处理前后关联的序列数据。

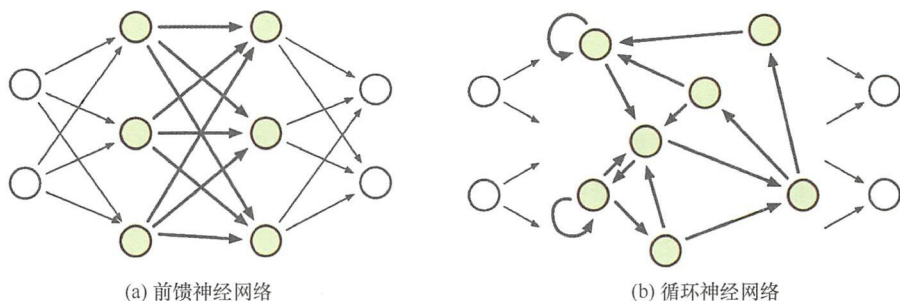


图 7-1 前馈神经网络与循环神经网络比较图

到目前为止, 我们只是初步认识了循环神经网络, 知道其能够处理序列数据, 但并没有了解其原理, 下面我们来深入了解循环神经网络。

### 7.1.1 前馈式神经网络的缺点

卷积神经网络是从人工神经网络逐渐发展而来的, 其特殊的卷积层在图像处理任务中具有出色的表现。但无论是人工神经网络, 还是卷积神经网络, 都存在着两个问题:

- **不能够对序列信号建模。**在传统的神经网络模型中, 神经元从输入层到隐层、再从隐层到输出层, 层层叠加, 层与层之间是全连接或者局部连接的方式。模型的前提是假定所有的输入和输出数据都是条件独立的, 因此有着固定的输入 (例如一张图片的大小、固定维度的向量特征) 和输出 (例如不同概率得分的分类)。例如, 上一张图像与下一张图像在时间上没有关联关系, 其标定分类在准备数据时已经固定下来。
- **单纯地前馈处理数据。**在神经网络的计算过程中, 没有记忆数据 (即没有存储计算过程中的数据), 该层网络计算完后传给下一层网络。此过程会丢失上一层计算过程产生的特征信息, 计算完的数据也没有向之前的信息进行反馈。

而循环神经网络之所以被称为“循环”, 是因为一个序列中的每一个元素都可能



是相关的，当前的输出依赖于之前的计算和新输入的数据，这就能够让循环神经网络对序列数据进行建模。我们也可以用记忆（Memory）来形容循环神经网络，其网络记住了到目前为止前面被计算过的信息，这样做就弥补了普通神经网络的缺点。

循环神经网络模型弥补了普通神经网络的缺点，能够对序列数据建模处理。下面来看什么是序列数据，有哪些数据是序列数据。

## 7.1.2 什么是序列数据

序列数据也可以被称为“序列信号”，而序列信号几乎无处不在，只要有先后关联关系或者时间关系的信号数据，都可以被认为是序列数据。

生活中最常见的例子就是文字，大部分人写汉字的顺序是从左到右、从上到下，后面一个词语依赖于前面词语的意思。我们需要根据前面出现的词语的意思，来预测句子中的后一个词语的意思。

例如：“打球才来下雨，希望在\_\_\_\_（昨天 / 今天 / 明天）”。该例子就是序列数据，根据前面词语出现的意思，可以推测后面下划线最有可能填的是“明天”。

另外一个与序列数据有关的很有意思的例子就是语音数据。【代码清单 7-1】为读取一段音频数据，并根据时间序列输出音频的波值。如图 7-2 所示，可以得知语音频率的高低代表不同词语的组合，一次高频接着低频很有可能为一个词语，后面将会出现的语音频率依赖于前面出现过的频率。因此，对于语音问题也可以使用循环神经网络模型进行建模。

【代码清单 7-1】读取音频数据并对音频数据进行可视化

```
from scipy.io import wavfile as wav
rate, data = wav.read('eastwood_lawyers.wav')
```

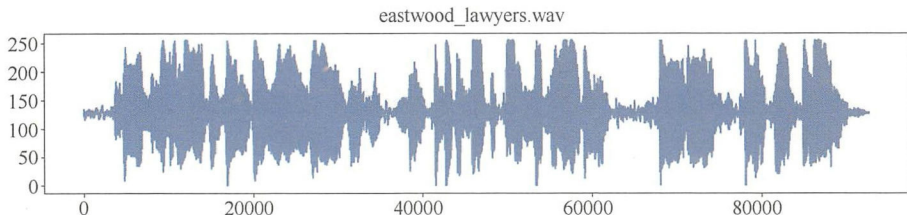


图 7-2 对语音序列信号进行可视化

更有趣的是一些看上去不是序列的数据，换个角度思考也可以当作序列问题来处理。如图 7-3 所示的手写字体“我”，因为文字笔画是有序的，因此可以根据已经出现的笔画特征，使用循环神经网络模型预测写的是什么字。





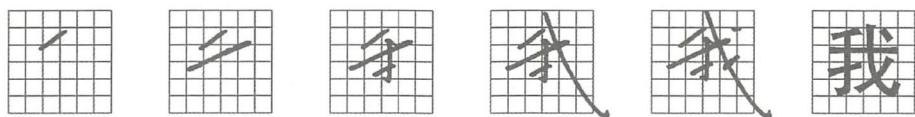


图 7-3 手写中文字体“我”

循环神经网络模型的目的就是对序列数据建模，而序列数据是与时间出现先后顺序有关联的数据。因为生活中绝大部分数据都是与时间有关联的，因此在深度学习中循环神经网络模型尤为重要。

## 7.2 循环神经网络的应用

在我们生活的时间和空间里，身边所发生的所有变化都可以使用序列数据来表示。如路由器根据访问网络的地址信息不断地调整自身所携带的信息；医生根据病人的病历数据和当前的身体状况，可以推测出该病人的病因，然后对症下药；淘宝会根据用户点击商品的顺序，推测出其可能购买的商品，进而推荐相应的商品广告等，上述都是应用序列数据的例子。正是因为序列数据无处不在，与我们的日常生活息息相关，所以对序列数据建模显得十分重要。

下面来了解循环神经网络模型的具体工程应用案例。

### 1. 语音识别 (Speech Recognition)

循环神经网络模型在语音识别中有着重大突破，如 (Graves et al., 2014) 使用双向循环神经网络模型输入音频数据，可以快速预测其对应的英文单词 (见图 7-4)，其准确率达到 92%。另外 (Huang et al., 2014) 使用双向循环神经网络模型实现单通道音乐的人声分离，实验结果表明该双向循环神经网络模型能够正确地单通道的歌曲中分离出人声和背景音乐，该技术可以应用在手机麦克风，在嘈杂环境下过滤掉背景噪声并提取出音频信号中的原声。

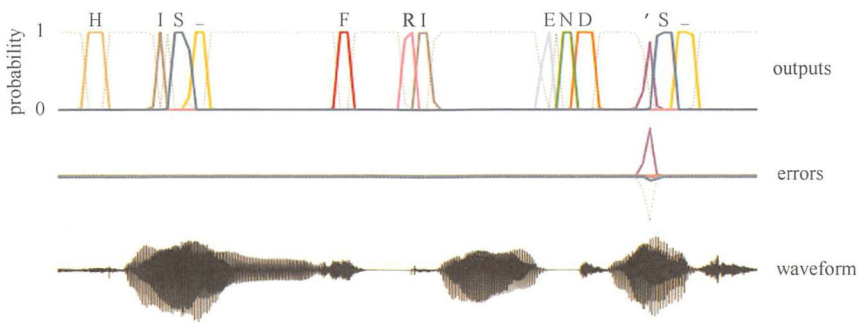


图 7-4 使用双向循环神经网络模型识别音频“HIS FRIENDS”





## 2. 图像识别 (Image Recognition)

(Wang et al., 2016) 利用循环神经网络模型解决跨年龄人脸识别问题。图 7-5 所示为改进循环神经网络模型基本结构的 Recurrent Face Aging (RFA) 网络, 该框架可以追踪 0 ~ 80 岁的人脸变化。

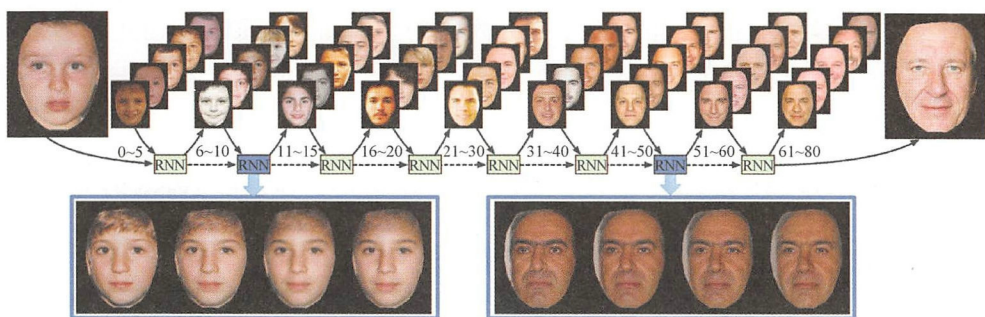


图 7-5 RFA 网络模拟人类年龄变化

在实际工程中, 由于缺乏一个人在不同年龄段带标签的人脸数据, 传统的人脸老化模型一般将年龄段分割成离散的年龄组。一个人在某个年龄时的人脸数据直接由其相邻的另一个不同年龄的数据经过特征变换获得。可是上述方法忽视了相邻年龄组之间固有的进化状态, 特征变换后得到的人脸往往具有重影尾迹。而人脸老化是一个平滑的过程, 所以更适合采用平滑变化的循环状态来估计人脸的年龄。

该思路与循环神经网络模型相结合, 通过使用改进后的循环神经网络模型, 模拟得到年龄缓慢变化的状态。该技术将来可以用在刑侦领域, 为国家公安体制提供强大的人工智能算法支撑。

## 3. 语言建模和文本生成 (Language Modeling and Generating Text)

循环神经网络模型在自然语言处理领域的应用则更为广泛, 诸如问答系统、情感分析、图像到文字的映射、机器翻译、语音识别、词性标注、命名实体识别等领域都有大量的进展。

其中最为著名的是在 2016 ~ 2017 年, Google 发布并升级的语言处理框架 SyntaxNet (见图 7-6), 一个用于分析和理解语法结构的循环神经网络模型, 在自然语言处理的语义理解领域的最新识别率上提高了 25%, 为 40 种语言带来了新的文本分割和词态分析功能。



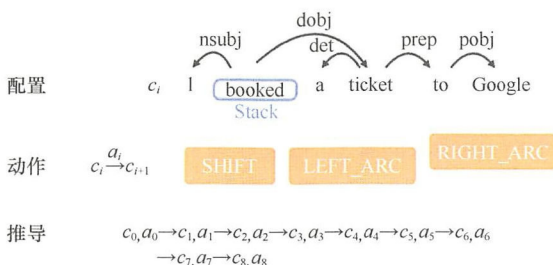


图 7-6 SyntaxNet 模型进行语义理解

## 7.3 循环神经网络的模型结构

### 7.3.1 序列数据建模

下面来看循环神经网络模型如何对序列数据进行建模。如图 7-7 (a) 所示, 在没有新输入的情况下, 序列数据在  $t$  时刻的状态是  $s_t$ , 该状态依赖于上一个时间点  $t-1$  的状态  $s_{t-1}$ 。依次类推, 下一个时间点  $t+1$  的状态  $s_{t+1}$  依赖于当前时间点  $t$  的状态  $s_t$ 。因此, 在没有新输入时, 当前序列状态的模型公式为:

$$s_t = f_{\theta}(s_{t-1}) \quad (7-1)$$

在正常情况下, 当前状态  $s_t$  不可能仅仅依赖于上一时刻的状态  $s_{t-1}$ , 否则从第一个时刻开始到序列终止时刻位置, 所有序列上的状态将会是一模一样的, 类似的序列数据模型就失去其意义。因此真实的情况是, 当前状态  $s_t$  依赖于上一时刻的状态  $s_{t-1}$  和当前的输入信息  $x_t$ , 当前时刻输入的信息对当前的状态  $s_t$  产生了影响。因此在带输入的情况下, 当前序列状态的模型公式为:

$$s_t = f_{\theta}(s_{t-1}, x_t) \quad (7-2)$$

如图 7-7 (b) 所示, 时刻  $t$  的状态  $s_t$  都是由当前输入  $x_t$  与上一时刻的状态  $s_{t-1}$  通过函数  $f_{\theta}$  计算得到的。

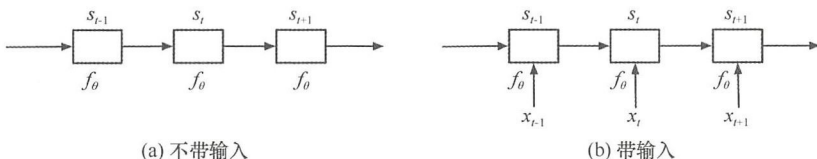


图 7-7 序列数据不带输入和带输入的情况



## 7.3.2 基本结构

那么有了当前状态和新输入信息，应在何时进行状态输出呢？如图 7-8 所示，左边是单个循环神经网络：由输入向量  $x$ 、隐层状态  $s$ 、输出向量  $h$  组成。其中隐层的输出有两个：一个输出反馈给自己，一个输出到下一时刻的神经元中。下面展开图 7-8 中的单个循环神经网络（图左半部分），得到图 7-8 右半部分所示的循环神经网络模型的基本结构。

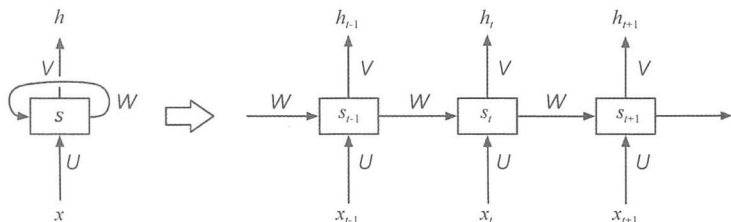


图 7-8 循环神经网络模型的基本结构

下面以图 7-8 为例，对循环神经网络模型的基本结构进行数学描述。

- $x_t$  :  $t$  时刻的输入， $X = [x_1, \dots, x_{t-1}, x_t, x_{t+1}, \dots, x_T]$  为输入序列；
- $s_t$  :  $t$  时刻的隐层状态（Hidden State），也被称为网络的记忆单元（Memory Unit），一个隐层状态可以包含多个神经元，其中  $S = [s_1, \dots, s_{t-1}, s_t, s_{t+1}, \dots, s_T]$ ；
- $h_t$  :  $t$  时刻的输出， $H = [h_1, \dots, h_{t-1}, h_t, h_{t+1}, \dots, h_T]$  为输出序列；
- $U$  : 输入序列信息  $X$  的权重参数矩阵；
- $W$  : 隐层状态  $S$  的权重参数矩阵；
- $V$  : 输出序列信息  $H$  的权重参数矩阵。

$s_t$  是根据上一个时刻  $t-1$  的隐层状态  $s_{t-1}$  和当前时刻  $t$  的输入  $x_t$  计算得到的。假设函数  $f$  为隐层状态的激活函数，当前隐层状态  $s_t$  的计算公式为：

$$s_t = f(Ws_{t-1}, Ux_t) \quad (7-3)$$

假设函数  $g$  为输出的激活函数，则输出的计算公式为：

$$h_t = g(Vs_t) \quad (7-4)$$

循环神经网络模型由式 (7-3) 和式 (7-4) 组成，与人工神经网络模型相比，循环神经网络模型参数多了  $W$  和  $V$  两个权重参数，并且网络数据传递方式也发生了改变。另外值得注意的是，图 7-8 的循环神经网络模型的基本结构中每一个时刻  $t$  都会对应一个输出  $h_t$ ，但在实际情况中，每一个时刻是否会有相对应的输出，需要根据具体任务需求而变化。例如，预测 Amazon 上用户对商品评论的好坏倾向，输出表示情



感倾向的分类，我们只在乎输入整句话之后的最终输出，而不是输入每一个单词后的输出，该情况下循环神经网络模型的输出只有 1 个。由此可以证明，循环神经网络模型的主要特性在于隐层状态  $s_t$  能够对序列数据具有记忆功能，通过隐层状态能够捕捉到序列信息之间的关系。

### 7.3.3 其他结构

循环神经网络结构模型并不只是图 7-8 所示中的一种，其网络模型结构经过组合排列可以有多种变化。图 7-9 所示均为单层循环神经网络模型，图中矩形框代表隐层状态，箭头则表示数据的流动方向。

- 在图 7-9 (a) 中，one to one 为单输入、单输出。因此单输入单输出的循环神经网络模型适用于词性分类、时序回归或者图像分类问题，例如输出该单词的词性。
- 在图 7-9 (b) 中，one to many 为单输入、序列输出。该循环神经网络模型用于图片字幕预测，如输入一张图片后输出一段文字序列；也可以作为解码器，如先训练好网络中的权重参数，给出一个单词解码一个句子。
- 在图 7-9 (c) 中，many to one 为序列输入、单输出。该循环神经网络模型可以用于文字情感分析，如输入一段文字，然后将其分为积极或者消极情绪，也可以作为的句子编码过程。
- 在图 7-9 (d) 中，many to many 为序列输入、序列输出。该循环神经网络模型可以用于机器翻译，如读入英文，语句然后将其以法语形式输出。
- 在图 7-9 (e) 中，many to many 为同步序列输入、序列输出。该循环神经网络模型适用于机器翻译模型、视频字幕翻译工具、机器问答系统等众多场合。

图 7-9 中的循环神经网络模型结构没有设置说明序列的长度，该长度与输入序列数据有关，具体需要根据实际工程的输入数据进行调整。

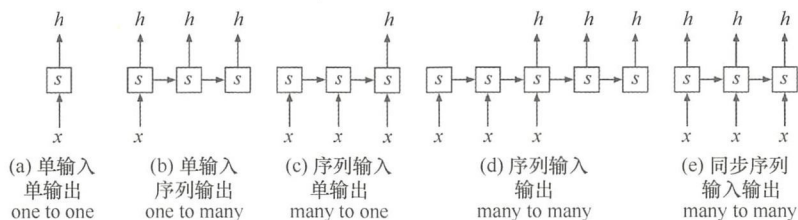


图 7-9 RNN 变种结构模型图





## 1. 双向循环神经网络 (Bidirectional Recurrent Neural Networks, BRNN)

时刻  $t$  的输出可能不仅依赖于序列前面的数据，也会依赖于后面将要输入的数据。例如：“打球才来下雨，\_\_\_\_\_（失望 / 希望 / 绝望）在明天”。遇到上述序列数据时，想要预测序列中下划线位置的单词，需要同时考虑单词前后的上下文关系（如下雨、明天），因此有了双向循环神经网络模型。

实际上，双向循环神经网络模型并不复杂，如图 7-10 所示，将两个循环神经网络模型的隐层状态  $s_t$  叠加并互相传播。

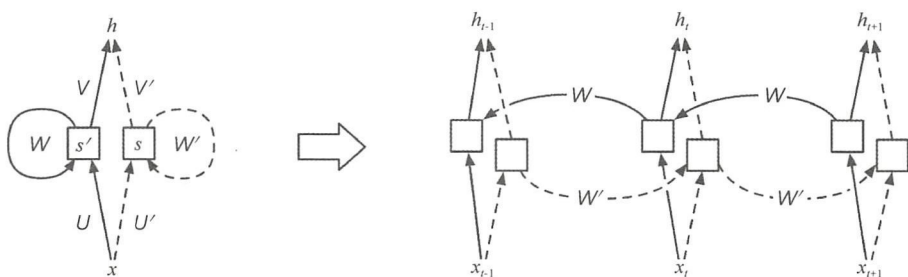


图 7-10 双向循环神经网络基本结构模型图

## 2. 深度循环神经网络 (Deep Recurrent Neural Networks, DRNN)

网络模型结构如图 7-11 所示。深度循环神经网络在基本循环神经网络结构的基础上进行改进，每一个时刻  $t$  对应多个隐层状态。该模型结构能够带来更好的学习能力，缺点在于难以对网络进行控制，并且随着网络层数的增多而引入更多的数学问题（例如梯度消失或者梯度爆炸等问题）。因此深度循环神经网络的模型结构对工程师提出了更高的要求，需要积累足够的网络训练与调参经验的工程师，才能设计出适合实际工程序列数据的循环神经网络模型。

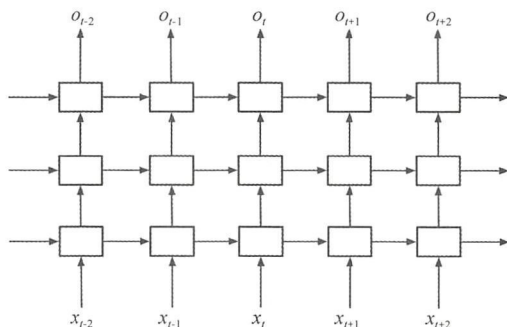


图 7-11 深度 RNN 基本模型结构



## 7.4 循环神经网络的核心算法

训练循环神经网络模型与训练传统的神经网络类似，仍然采用随机梯度下降算法（SGD）和反向传播算法（BP），但是在计算细节方面进行了部分修改。在循环神经网络模型中，其权重参数在不同时序上参数共享，每个节点参数梯度不但依赖于当前时间步的计算结果，同时还依赖于上一时间步的计算结果。例如，为了计算时刻  $t = 4$  的梯度，我们需要使用 3 次反向传播算法，为了改进计算反向传播，通过时序的方法被称为时间反向传播算法（Backpropagation Through Time, BPTT）。在本节中，我们将会详细介绍循环神经网络模型，以及通过 BPTT 算法计算循环神经网络模型损失函数关于多个权重参数的梯度。

### 7.4.1 模型详解

在上一节中，我们初步了解了循环神经网络的基本结构。实际的循环神经网络模型如图 7-12 所示， $t$  时刻的输入向量  $x_t$  由多个神经元组成，每一个神经元对应多维向量中的一个值。隐层状态  $s_t$  作为一个记忆单元，包含有  $D_h$  个神经元，隐层状态  $s_t$  之间使用权重矩阵  $W$  进行连接。通过权重矩阵  $U$ ，可以将输入向量与隐层状态中的神经元连接。同理，隐层状态的内部神经元通过权重矩阵  $V$  与输出向量中的神经元相连接，最终形成一个完整的循环神经网络模型。

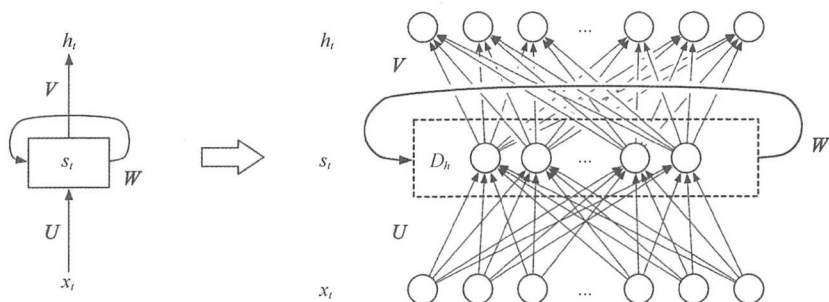


图 7-12 循环神经网络的基本结构详细展开图。图中左半部分为循环神经网络结构的简单示例，右半部分为循环神经网络结构的详细示例

图 7-13 所示为循环神经网络模型的简单结构展开，假设其隐层状态中激活函数使用 Tanh 函数，输出层中激活函数使用 Softmax 函数，则对应的隐层状态  $s_t$  和输出  $h_t$  有：



$$s_t = \tanh(Ux_t + Ws_{t-1}) \quad (7-5)$$

$$h_t = \text{softmax}(Vs_t) \quad (7-6)$$

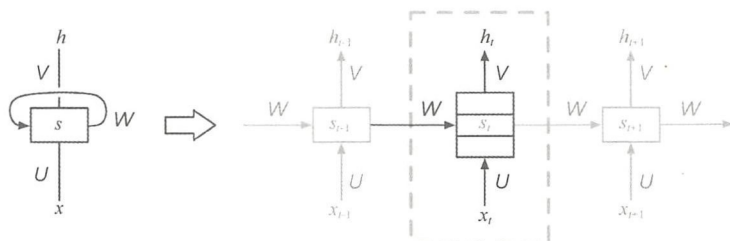


图 7-13 单层 RNN 网络结构展开图，其中隐层状态  $s_t$  有 3 个神经元 ( $D_h=3$ )

已知隐层状态和输出的计算公式，接着需要确认网络中的权重矩阵  $U$ 、 $W$  和  $V$ 。假设在时间步  $x_t$  的向量长度  $d$  为 1000，对应时间步中隐层状态  $s_t$  的神经元数  $D_h$  为 50，那么根据循环神经网络模型基本结构定义有：

$$\begin{aligned} x_t &\in \mathbf{R}^d \rightarrow x_t \in \mathbf{R}^{1000} \\ h_t &\in \mathbf{R}^d \rightarrow h_t \in \mathbf{R}^{1000} \\ s_t &\in \mathbf{R}^{D_h} \rightarrow s_t \in \mathbf{R}^{50} \\ U &\in \mathbf{R}^{D_h \times d} \rightarrow U \in \mathbf{R}^{50 \times 1000} \\ W &\in \mathbf{R}^{D_h \times D_h} \rightarrow W \in \mathbf{R}^{50 \times 50} \\ V &\in \mathbf{R}^{d \times D_h} \rightarrow V \in \mathbf{R}^{1000 \times 50} \end{aligned}$$

现在我们确定了循环神经网络模型输入层、隐层、输出层的参数矩阵大小，还确定了网络中的权重矩阵  $U$ 、 $W$  和  $V$  的大小，即可知道执行循环神经网络模型大概需要多少参数。其中权重矩阵  $U$ 、 $W$ 、 $V$  的大小分别为  $D_h \times d$ 、 $D_h \times D_h$ 、 $d \times D_h$ ，因此循环网络中的总权重参数大小约为  $2D_h \times d + D_h^2$ ，按照计算，一共需要约 102500 个权重参数。相对而言循环神经网络模型的权重参数远远少于卷积神经网络模型所需要的权重参数，模型的权重参数不大，泛化能力不足都是制约循环神经网络性能的因素（另外一个因素是梯度问题，将在后续章节中介绍）。

值得一提的是，隐层状态中的神经元  $D_h$  可以认为是循环神经网络的记忆能力神经元，隐层内的神经元越多，可以学习越复杂的模式（模拟更复杂的函数），但这并不意味着隐层神经元越多越好。网络模型中的神经元过多，可能会引入过度拟合等问题。

有了循环神经网络结构中各个参数的大小定义后，现在可以初始化网络权重参数。根据第 3 章中深度学习技巧有关网络参数的初始化的内容，当网络权重的参数



初始化为标准差 $1/\sqrt{n}$ 的高斯随机分布时，训练效果较佳。

因此循环神经网络模型中权重矩阵 $U$ 、 $W$ 、 $V$ 使用标准差为 $1/\sqrt{n}$ 、均值为0的高斯随机分布函数初始化每一个权重参数。【代码清单 7-2】为对循环神经网络模型的权重参数进行初始化，其中 input\_dim 为假设在时间步 $x_t$ 的向量长度 $d$ ，hidden\_dim 为对应时间步中隐层状态 $s_t$ 的神经元数 $D_h$ 。

【代码清单 7-2】初始化循环神经网络模型

```
class SmapleRNN:
    def __init__(self, input_dim, hidden_dim=50):
        # 设置 RNN 网络的输入、输出、隐层的维度
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim

        # 定义 RNN 网络权重大小
        U_size = (self.hidden_dim, self.input_dim)
        W_size = (self.hidden_dim, self.hidden_dim)
        V_size = (self.input_dim, self.hidden_dim)

        # 随机初始化 RNN 网络权重参数
        random_min = -np.sqrt(1./self.input_dim)
        random_max = np.sqrt(1./self.input_dim)
        self.U = np.random.uniform(random_min, random_max, U_size)
        self.W = np.random.uniform(random_min, random_max, W_size)
        self.V = np.random.uniform(random_min, random_max, V_size)

    def _softmax(self, x):
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum(axis=0)

    def _tanh(self, x):
        return np.tanh(x)
```

## 7.4.2 向前传播

序列数据作为循环神经网络模型的输入，根据序列的先后顺序向前传播，具体过程如图 7-14 所示，每一个时间序列需要计算一次隐层状态 $s_t$ 和输出 $h_t$ 的值。从第一个时间序列 $t=0$ 开始，计算 $s_0$ 和输出 $h_0$ ；接着计算第二个时间序列 $t=1$ ，第一个时间序列的隐层状态 $s_0$ 和序列数据 $x_1$ 作为网络输入，输出 $s_1$ 和 $h_1$ 。依此类推，直到最后一个时间序列 $T$ 。





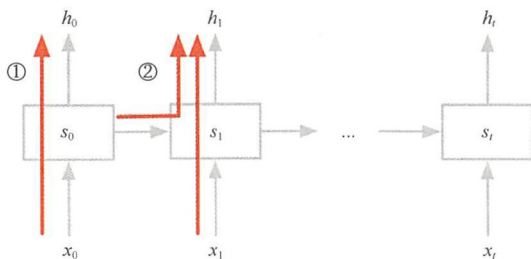


图 7-14 循环神经网络模型向前传播过程。①为第 1 个时刻向前传播，②为第 2 个时刻向前传播

【代码清单 7-3】中的 `forward_propagation()` 函数实现循环神经网络模型的向前传播功能，并返回当前时间步  $t$  的输出值  $h_t$  和隐层状态  $s_t$ 。其中， $h[t]$  表示在时间步  $t$  的输出。

在输出预测阶段，我们总希望序列中的下一个单词应该获得最高的概率，因此预测函数 `predict()` 通过 `np.argmax` 提取每一个输出向量的最大值所在的位置索引，作为该序列的实际输出。

#### 【代码清单 7-3】向前传播算法

```
def forward_propagation(self, x):
    """ 向前传播算法 """
    time = x.shape[0] # 获取输入数据中的序列数 time

    s = np.zeros((time+1, self.hidden_dim)) # 定义隐层状态矩阵
    h = np.zeros((time, self.input_dim)) # 定义输出

    # 根据公式计算隐层状态和输出的值
    for t in np.arange(time):
        s[t] = self._tanh(np.dot(self.U, x[t]) + np.dot(self.W, s[t-1]))
        h[t] = self._softmax(self.V.dot(s[t]))

    return (s, h)

def predict(self, x):
    """ 根据输入序列数据预测序列输出 """
    s, h = self.forward_propagation(x)

    print("input shape:{}.".format(x.shape))
    print("status shape:{}.".format(s.shape))
    print("output shape:{}.".format(h.shape))

    return np.argmax(h, axis=1)
```

`SmapleRNN.forward_propagation = forward_propagation`



### 7.4.3 损失函数

为了训练循环神经网络模型，需要找到一种方法来度量循环神经网络模型输出产生的误差，该方法同样是损失函数（Loss Function）。实际上循环神经网络模型的损失函数的定义与第3章所介绍的并没有太大区别。循环神经网络模型常用的损失函数为交叉熵函数，其中 $y$ 是真实输出值， $\hat{y}$ 是神经网络预测的输出值：

$$L(y, \hat{y}) = - \sum_x y(x) \log \hat{y}(x) \quad (7-7)$$

我们的目标是优化循环神经网络中的权重参数矩阵 $U$ 、 $W$ 、 $V$ ，使得输入的序列数据经过循环神经网络后的输出值更加接近真实的输出值。也就是说，希望找到能够在给定训练数据的情况下，使损失函数最小化的一组参数。 $y$ （标定真实值）与 $\hat{y}$ （网络预测值）之间相差越大，意味着网络模型的误差越大，其损失也就越大。

假设在循环神经网络中时间步 $t$ 的损失函数为 $L^t$ ，则有：

$$L^t = L^t(y, \hat{y}) = -y_t \log \hat{y}_t \quad (7-8)$$

如果输出的时间序列数为 $T$ ，那么循环神经网络模型的总损失函数为：

$$L = \sum_t^T L^t = \sum_t^T -y_t \log \hat{y}_t \quad (7-9)$$

在【代码清单 7-4】中，函数`calc_loss()`实现循环神经网络的损失函数计算。首先通过一次向前传播得到循环神经网络预测的输出 $\hat{y}$ (`y_predict`)，接着计算时刻 $t$ 的损失函数 $L^t$ ，最后求出总损失 $L$ 。代码中使用矩阵操作，方便一次性计算所有时间序列的损失。

【代码清单 7-4】循环神经网络模型损失函数

```
def calc_loss(self, x, y):  
    """ 计算损失 """  
    loss = 0  
    time = y.shape[0]  
  
    s, y_predict = self.forward_propagation(x)  
  
    # t 时间上的损失  
    loss_t = y.T * np.log(y_predict)  
  
    total_loss = - np.sum(loss_t)  
    return total_loss
```

```
SmplerRNN.calc_loss = calc_loss
```



## 7.4.4 时间反向传播算法

有了循环神经网络模型后，需要对比网络的输出预测值和真实输出值之间的差异，循环神经网络通过时间反向传播（BPTT）算法对损失函数进行求导，获得网络中所有参数的梯度。在循环神经网络模型的训练过程中，同样使用到了随机梯度下降（SGD）算法，迭代地调用 BPTT 算法求得网络参数梯度。在迭代过程中，通过学习率轻轻地推动网络参数朝着误差减少的方向去改变，从而更新循环神经网络模型中的权重参数矩阵  $U$ 、 $W$ 、 $V$ 。

这就是循环神经网络确定权重参数的方式，那么 BPTT 算法在时间序列上是如何对损失函数进行求导的呢？

由于循环神经网络是基于时序数据的神经网络模型，因此传统的反向传播算法并不适用于循环神经网络模型的参数训练。在循环神经网络模型中，最常用的是通过 BPTT 算法求得网络中的权重参数  $U$ 、 $W$ 、 $V$  的导数。值得注意的是，循环神经网络模型中的权重参数  $U$ 、 $W$ 、 $V$  在网络的所有时间步中通过共享使用，所以每一个时间步的输出梯度不仅依赖于当前时间步  $t$  的运算，也包括了上一时间步  $t-1$  的运算。

通过 BPTT 算法，输入训练样本  $(x_t, y_t)$ ，可以得到权重参数的导数  $\frac{\partial L}{\partial V}$ 、 $\frac{\partial L}{\partial W}$ 、 $\frac{\partial L}{\partial U}$ 。

实际上，BPTT 算法与 BP 算法类似，只是多了在时间上反向传递的梯度。我们的目标是求得导数  $\frac{\partial L}{\partial U}$ 、 $\frac{\partial L}{\partial W}$ 、 $\frac{\partial L}{\partial V}$ ，根据网络中 3 个权重参数的变化率来优化网络参数  $U$ 、 $W$ 、 $V$ 。因为有：

$$\frac{\partial L}{\partial U} = \sum_t \frac{\partial L_t}{\partial U}, \quad \frac{\partial L}{\partial W} = \sum_t \frac{\partial L_t}{\partial W}, \quad \frac{\partial L}{\partial V} = \sum_t \frac{\partial L_t}{\partial V} \quad (7-10)$$

因此我们只需要对每个时刻的损失函数求偏导，得到该时刻损失函数关于权重参数的导数，再进行相加即可得到总的导数（如图 7-15 所示）。

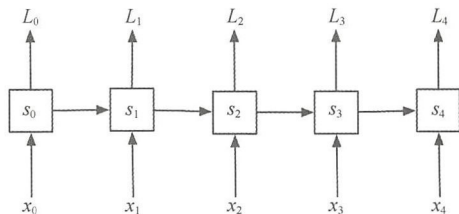


图 7-15 在 BPTT 时间反向传播算法中算出每一个时刻  $t$  对应的损失函数  $L_t$ ，网络总损失就是对每一个时刻的损失求和

如果读者不想深入了解 BPTT 算法，可以直接跳过本节的公式推导部分，直接



使用推导结论或查阅对应的 Python 代码。

在正式进入 BPTT 时间反向传播算法之前，我们先回顾图 7-16 所示对应的循环神经网络模型的基本公式：

$$z_t = Ux_t + Ws_{t-1} \quad (7-11)$$

$$s_t = \tanh(z_t) \quad (7-12)$$

$$\hat{y}_t = \text{softmax}(Vs_t) \quad (7-13)$$

$$L_t = -y_t \log \hat{y}_t \quad (7-14)$$

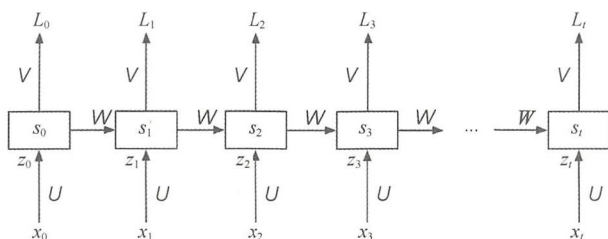


图 7-16 循环神经网络基本公式对应网络中的参数位置

为了计算循环神经网络模型中权重参数  $U$ 、 $W$ 、 $V$  的导数，我们使用 BP 算法中的导数链式法则，从损失函数的误差反向开始。

### 1. 权重 $V$ 的梯度

复合矩阵函数  $G = G(X)$  求导法则为：

$$G = G(X) \rightarrow \frac{\partial g(G)}{\partial X} = \text{tr} \left( \left( \frac{\partial g(G)}{\partial G} \right)^T \frac{\partial G}{\partial X} \right) \quad (7-15)$$

其中， $\text{tr}$  是矩阵的迹。在线性代数中， $n \times n$  的对角矩阵  $A$  的主对角线上各个元素的总和被称为矩阵  $A$  的迹。于是根据式 (7-15)，在时刻  $t$  损失函数对  $V$  进行求导有：

$$\frac{\partial L_t}{\partial V} = \text{tr} \left( \left( \frac{\partial E_t}{\partial \hat{y}_t} \right)^T \frac{\partial \hat{y}_t}{\partial V} \right) = \text{tr}((\hat{y}_t - y_t)^T s_t) \quad (7-16)$$

下面对  $V$  矩阵的每一个位置进行求导计算，其中  $(\hat{y}_t - y_t)^{(i)}$  表示向量中的第  $i$  个元素， $s_t^{(j)}$  表示隐层神经  $s_t$  向量第  $j$  个元素，这时候求向量的迹就变成求向量外积：

$$\frac{\partial L_t}{\partial V_{ij}} = \text{tr}((\hat{y}_t - y_t)^{(i)} s_t^{(j)}) = (\hat{y}_t - y_t) \otimes s_t \quad (7-17)$$





以时间步  $t=3$  的损失值  $L_3$  为例,  $V$  的梯度为:

$$\frac{\partial L_3}{\partial V} = (\hat{y}_3 - y_3) \otimes s_3 \quad (7-18)$$

从式 (7-14) 中可以看出,  $\frac{\partial L_3}{\partial V}$  的值只依赖于当前时间步  $t$ , 有了与当前时间步相关的 3 个值  $\hat{y}_3$ 、 $y_3$ 、 $s_3$ , 就可以使用简单的矩阵操作计算出网络权重向量  $V$  的梯度。

【代码清单 7-5】中首先向前传播求得所有时间步的状态和输出预测值, 然后向后迭代求每一个时间步中  $V$  的梯度。

#### 【代码清单 7-5】计算权重参数 $V$ 的梯度

```
# 首先通过向前传播求得所有时间步的状态和预测
s, y_predict = SampleRNN.forward_propagation(x)

# 残差向量
dE_dy = y - y_predict

# 向后迭代求 v 的梯度
# 使用 reversed 对 range 求得的数组 [0, 1, 2, ..., time] 进行反转
for t in reversed(range(time)):
    dE_dV += np.outer(dE_dy[t], s[t].T) # 实现公式 (7-13)
```

## 2. 权重 $W$ 的梯度

权重  $W$  在隐层状态所有时间步上进行共享, 在时间步  $t$  之前每一个时刻  $W$  的变化都对损失值  $L_t$  产生影响, 因此在反向求导时, 也需要考虑之前每一个时刻  $t$  上  $W$  对  $L$  的影响, 著名的时间反向传播算法就是由此而来的。

由于有  $s_t = \tanh(Ux_t + Ws_{t-1})$ , 同样以时间步  $t=3$  为例 (如图 7-17 所示),  $s_3 = \tanh(Ux_3 + Ws_2)$  依赖于  $s_2$ , 而  $s_2 = \tanh(Ux_2 + Ws_1)$  则依赖于  $s_1$ , 依次类推直到  $s_0$ 。所以想要对  $W$  求导, 需要使用链式法则:

$$\frac{\partial L_t}{\partial W} = \sum_{k=0}^t \frac{\partial L_t}{\partial s_k} \frac{\partial s_k}{\partial W} \quad (7-19)$$

以  $L_3$  为例, 如图 7-18 所示,  $L_3$  求导为 4 个反向求导公式的和, 每一条链式法则都加入前一个时间步进行计算。因为参数  $\frac{\partial s_3}{\partial s_2}$ 、 $\frac{\partial s_2}{\partial s_1}$ 、 $\frac{\partial s_1}{\partial s_0}$  是重复计算的, 因此在实际的计算过程中上述参数可以进行共享。

对应图 7-18, 我们对时间步 3 的损失函数关于权重参数  $W$  的导数  $\frac{\partial L_3}{\partial W}$  进行展开得到:



$$\begin{aligned}
 \frac{\partial L_3}{\partial W} &= \sum_{k=0}^3 \frac{\partial L_3}{\partial s_k} \frac{\partial s_k}{\partial W} \\
 &= \underbrace{\frac{\partial L_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0} \frac{\partial s_0}{\partial W}}_{\textcircled{1}} + \underbrace{\frac{\partial L_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial W}}_{\textcircled{2}} + \underbrace{\frac{\partial L_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial W}}_{\textcircled{3}} + \underbrace{\frac{\partial L_3}{\partial s_3} \frac{\partial s_3}{\partial W}}_{\textcircled{4}} \quad (7-20)
 \end{aligned}$$

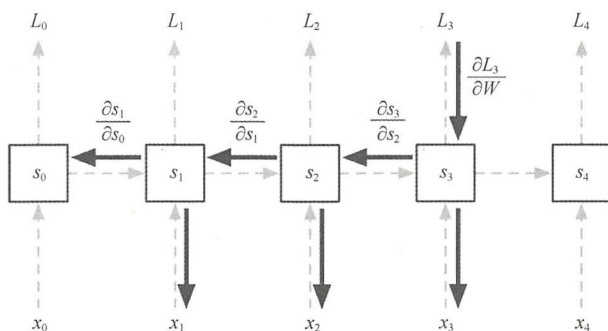
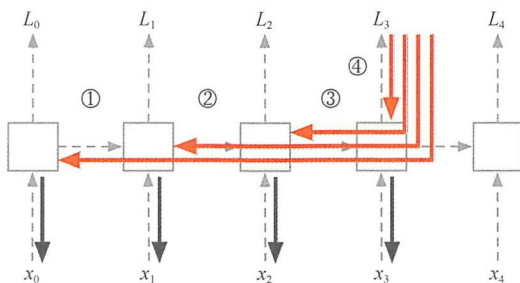


图 7-17 时间反向传播算法中各时间段的导数

图 7-18 对权重参数  $W$  使用链式法则求导

最后根据复合矩阵函数  $G = G(X)$  求导法则来求导，把  $\delta_k = \frac{\partial L_t}{\partial s_k}$  代入式 (7-20)

得到：

$$\frac{\partial L_t}{\partial W} = \sum_{k=0}^t \frac{\partial L_t}{\partial s_k} \frac{\partial s_k}{\partial W} = \sum_{k=0}^t \text{tr} \left[ \left( \frac{\partial L_t}{\partial s_k} \right)^T \frac{\partial s_k}{\partial W} \right] = \sum_{k=0}^t \text{tr} \left[ (\delta_k)^T \frac{\partial s_k}{\partial W} \right] \quad (7-21)$$

其中， $\delta_t$  和  $\delta_k$  应用链式法则为：



$$\delta_t = \frac{\partial L_t}{\partial z_t} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial s_t} \frac{\partial s_t}{\partial z_t} = (V^T (\hat{y}_t - y_t)) \odot (1 - s_t^2) \quad (7-22)$$

$$\delta_k = \frac{\partial L_t}{\partial z_k} = \frac{\partial L_t}{\partial s_{k+1}} \frac{\partial s_{k+1}}{\partial s_k} \frac{\partial s_k}{\partial z_k} = (W^T \delta_{k+1}) \odot (1 - s_t^2) \quad (7-23)$$

与求  $V$  的梯度一样使用矩阵形式表达, 可以得到:

$$\frac{\partial L_t}{\partial W} = \sum_{k=1}^t \delta_k \otimes s_{k-1} \quad (7-24)$$

以  $L_3$  为例, 其关于权重参数  $W$  的导数  $\frac{\partial L_3}{\partial W}$  为:

$$\frac{\partial L_3}{\partial W} = \sum_{k=0}^3 \delta_k \otimes s_{k-1} = \delta_3 \otimes s_2 + \delta_2 \otimes s_1 + \delta_1 \otimes s_0 \quad (7-25)$$

【代码清单 7-6】是 BPTT 算法的实现。设置 `bptt_truncate` 为 4, `bptt_truncate` 作用于 for 循环中的迭代次数, 限制 BPTT 向后计算参数  $W$  和  $U$  的更新次数。限制其计算是因为在实际生产环境中, 标准的循环神经网络模型是很难进行训练的。序列越长, 反向传播越久, 其计算量越大, 而且也很容易引起梯度消失或者梯度爆炸问题。因此在实践中, 常把时间反向传播算法的时间序列计算控制在向后 `bptt_truncate` 个时间步内。

#### 【代码清单 7-6】BPTT 算法实现

```
def backward_propagation_through_time(self, x, y):
    bptt_truncate = 4 # 反向传播截断参数

    time = y.shape[0] # 输出的序列数

    # 首先通过向前传播求得所有时间步的状态和预测
    s, y_predict = SampleRNN.forward_propagation(x)

    # 初始化权重矩阵的梯度
    dE_dV = np.zeros_like(self.V)
    dE_dW = np.zeros_like(self.W)
    dE_dU = np.zeros_like(self.U)

    dE_dy = y - y_predict

    for t in reversed(range(time)):
        dE_dV += np.outer(dE_dy[t], s[t].T)

    # 开始计算 w 和 u 的梯度
```



```

# 首先计算 delta t, 在第一次计算的时候 t = 3
delta_k = (self.V * dE_dy) * (1 - pow(s[t], 2))

# 开始使用 BPTT 进行链式法则求导
for step_t in reversed(arange(t + 1)):
    dE_dW += np.outer(delta_k, s[step_t-1]) # 加到之前每一步的梯度上
    dE_dU[:, x[step_t]] += delta_k
    delta_k = (self.W * delta_k) * (1 - pow(s[step_t-1], 2))

```

BP 算法只考虑了上下层之间梯度的纵向传播, BPTT 算法同时考虑了层级间的纵向传播和时间上的横向传播, 并同时在时间序列和当前层神经元传递两个方向上进行参数优化。

## 7.4.5 梯度消失与梯度爆炸

### 1. 梯度问题

循环神经网络模型的设计初衷是处理序列数据。假设有例句“她是一个经常出现在我梦中, 美丽、漂亮、大方、得体, 看上去又有点文雅的女人”, 想使用循环神经网络模型来提取例句中的句意“她是女人”, 效果却难以达到我们的预期, 因为循环神经网络模型很难学习到长期依赖的序列数据中的有效数据。求时间步  $t$  隐层的权参数为:

$$\frac{\partial L_t}{\partial W} = \sum_{k=0}^t \frac{\partial L_t}{\partial s_k} \frac{\partial s_k}{\partial W} \quad (7-26)$$

式 (7-26) 中  $\frac{\partial L_t}{\partial s_k}$  的求解使用链式法则展开, 可以改写为:

$$\frac{\partial L_t}{\partial W} = \sum_{k=0}^t \frac{\partial L_t}{\partial s_k} \frac{\partial s_k}{\partial W} = \sum_{k=0}^t \frac{\partial L_t}{\partial s_k} \left( \prod_{j=k+1}^t \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W} \quad (7-27)$$

其中激活函数  $s_t = f(Ux_t + Ws_{t-1})$ , 当选择 Tanh 作为激活函数时, 其导数取值范围如图 7-19 (a) 所示为  $[-1,1]$ ; 选择 Sigmoid 函数作为激活函数时, 其导数取值范围为  $[0,0.25]$ 。网络序列越长, 根据链式法则, 导数相乘越多, 会使得式 (7-27) 趋近于零。换句话说, 多个小于 1 的项连成结果很快会逼近于零, 从而引起梯度消失问题。常见的激活函数及其导数见【代码清单 7-7】。

【代码清单 7-7】常用的激活函数及其导数

```

def tanh(x):
    return np.tanh(x)

```





```
def tanh_deriv(x):  
    return 1.0 - np.tanh(x)**2  
  
def sigmoid(x):  
    return 1 / (1 + np.exp(-x))  
  
def sigmoid_deriv(x):  
    return sigmoid(x) * (1 - sigmoid(x))  
  
def relu(data, epsilon=0.1):  
    return np.maximum(epsilon * data, data)  
  
def relu_deriv(data, epsilon=0.1):  
    gradients = 1. * (data > 0)  
    gradients[gradients == 0] = epsilon  
    return gradients
```

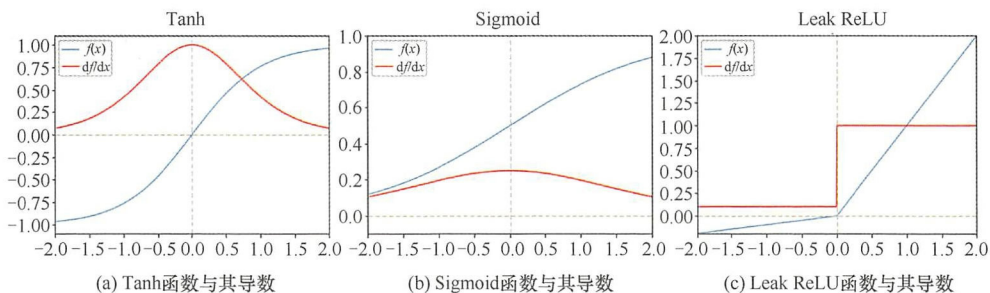


图 7-19 Tanh 函数、Sigmoid 函数、Leak ReLU 函数及其对应的导数

如果发生梯度消失问题，则意味着距离当前时刻非常远的输入数据不能为当前时刻的模型参数的更新做出贡献。如果距离当前步很远的时间步对梯度的贡献为零，则意味着往后的时间步上状态都对学习过程没有任何帮助。这表明我们在还没学习到长依赖序列数据时，循环神经网络模型就已经结束训练了。

## 2. 避免梯度问题

由于梯度消失和梯度爆炸问题的出现，即使是使用 BPTT 训练一个最简单的循环神经网络模型，也很难解决时序上长距离依赖的问题。常用的避免或者解决梯度消失和梯度爆炸问题的方法有以下 5 种。

### (1) 截断梯度

截断梯度是在循环神经网络更新参数时，只利用较近时刻的序列信息，而忽略历史悠久的信息。在【代码清单 7-8】中，BPTT 时间反向传播算法中设置了一个



bptt\_truncate=4, 该参数就是使用截断梯度的思想, 用于限制 BPTT 算法向后传播的次数, 减少长依赖的问题。

### 【代码清单 7-8】阶段梯度实现

```
def backward_propagation_through_time(self, x, y):
    bptt_truncate=4

    ...
    # 开始使用 BPTT 进行链式法则求导
    for step_t in reversed(arange(t - bptt_truncate, t + 1)):
        dE_dW += np.outer(delta_k, s[step_t-1]) # 加到之前每一步的梯度上
        dE_dU[:, x[step_t]] += delta_k
        delta_k = (self.W * delta_k) * (1 - pow(s[step_t-1], 2))
```

### (2) 设置梯度阈值

在梯度爆炸时, 程序会检测到梯度数值很大或者溢出, 因此可以设置一个梯度阈值, 在梯度超过阈值之后直接截断。

下面是梯度截断的伪代码, 首先获得该梯度, 接着判断该梯度的值是否大于阈值。如果大于, 则根据公式更新梯度, 否则输出原来的梯度。

$$\text{BEGIN: } \delta \leftarrow \frac{\partial L}{\partial s}$$

$$\text{If } \|\hat{\delta}\| \geq \text{threshold then } \hat{\delta} \leftarrow \frac{\text{threshold}}{\|\hat{\delta}\|} \hat{\delta}$$

End if

在【代码清单 7-9】中, gradients\_clipping() 函数为对梯度阈值伪代码的 Python 实现。

### 【代码清单 7-9】梯度截断函数

```
def gradients_clipping(g, th = 0.8):
    """ 梯度截断: 根据给定的阈值在每一次求得导数后进行判别 """
    if (g > th):
        g = th / abs(g) * g # abs(g) 取导数的绝对值
    return g
```

### (3) 合理初始化权重值

合理地初始化权重值, 让循环神经网络模型中每个神经元尽可能不要取极大值或极小值, 以避免可能导致梯度消失的区域。例如在使用高斯概率分布得到初始化权重值后, 对接近极值的数进行修正, 让其更集中在分布中心, 或者使用预训练的网络。



#### (4) 使用 ReLU 作为激活函数

使用 ReLU 代替 Sigmoid 和 Tanh 作为激活函数。如图 7-19 (c) 所示, ReLU 的导数被限制成 0 或者 1, 因此更能够容忍梯度扩散或者梯度消失问题。

#### (5) 使用 LSTM 或者 GRU 作为记忆单元

解决梯度问题最流行的解决方法就是使用长短期记忆 (Long Short-Term Memory, LSTM) 或者门控循环单元 (Gated Recurrent Unit, GRU) 结构, 代替原神经网络模型中的记忆单元。其中, LSTM 早在 1997 年就被第一次提出, 后经过 (Alex et al., 2014) 改进而成为了近年来处理自然语言最为广泛的网络模型之一; 而 GRU 则是在 2014 年被首次提出的。两种循环神经网络的结构都经过了精心设计, 被用来解决梯度扩散和长期依赖问题。

## 7.5 示例：使用循环神经网络预测文本数据

### 7.5.1 定义网络模型

给定一个句子, 我们可以将该句子以概率形式表达:

$$P(AB) = P(B)P(A|B) \quad (7-28)$$

例如句子 “the clouds are in the sky.” ( $AB$ ) 是在 “sky” ( $B$ ) 与 “the clouds are in the” ( $A|B$ ) 条件下的概率相乘。同理有 “the clouds are in the” 的概率是 “the” 与 “the clouds are in” 条件下的概率相乘, 依此类推直到句子开头。反过来, 我们按时间序列输入单词, 根据概率条件预测一个可能出现单词的概率, 不断重复该过程, 最终可以获得一个完整的句子。

有的读者会有疑问, 获得一个完整的句子有什么用呢? 或者说为什么想要知道未来单词出现的概率呢? 这就是循环神经网络模型的魅力所在。假设在使用搜索引擎时, 用户只输入了部分搜索单词, 在没有输入完数据之前, 搜索引擎就能提示我们最有可能搜索的是什么 (如图 7-20 所示)。在使用手机语音提醒功能时, 当我们还没有把一句话完整说完之前, 语音助手就可以帮我们自动校正和提醒可能说的话, 这将会极大地方便用户使用。循环神经网络模型构建的时序任务模型可以有更广阔的应用空间, 让深度学习真正地服务人类。

本节将会通过实际的例子来学习如何构建循环神经网络的基本模型。以自然语言处理任务中对句子进行单词预测作为背景, 根据当前输入的句子序列数据, 预测下一个可能出现的单词。



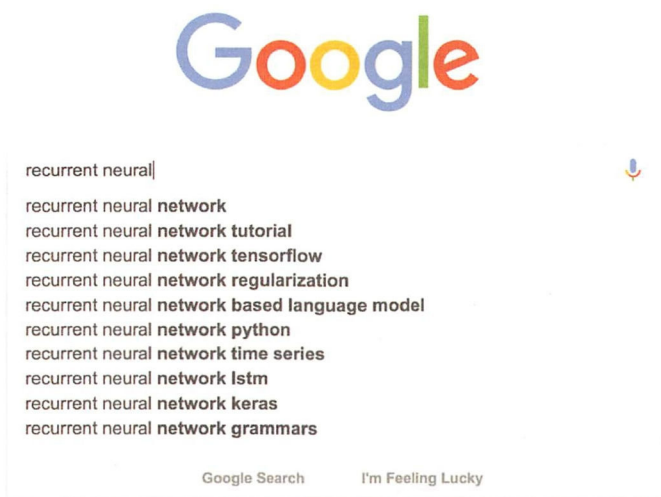


图 7-20 Google 搜索栏，分析句子成分并给出下一个单词的预测

## 7.5.2 序列数据预处理

我们的任务是基于每个单词的输入时序，预测下一个可能出现的单词。以 “the clouds are in the sky.” 为例，假设已经知道第一个单词 “the”，需要预测下一个单词为 “clouds” 的概率；同理，当知道句子的前 5 个单词 “the clouds are in the”，需要预测下一个单词出现 “sky” 的概率。

既然是以单词为最小单位进行预测，那么就需要把文本数据切分成句子，把句子切分成组成该句子的单词。在这里使用 NLTK 工具包的 `word_tokenize` 把句子 “the clouds are in the sky.” 切分成 7 个组成成分 (“the", "clouds", "are", "in", "the", "sky", "."), 每个成分称为 Token。

在对文本序列数据预处理时，我们需要注意几个方面。词表字典越大，意味着句子向量矩阵越大，这无疑会增加模型的训练时间和训练难度。因此对于文本当中的一些稀有词汇，应该将这些不常见的单词删掉，可以使用 “UNKNOWN” 来代替这些稀有词汇。例如词表字典中没有 “sky” 这个单词，那么句子 “the clouds are in the sky.” 将会转换成 ("the", "clouds", "are", "in", "the", "UNKNOWN", ".")。值得注意的是，UNKNOWN 需要作为一个独立的单词放入词表字典中。

除了稀有词汇，还应该标记句子的起始位置和结束位置，因此会在每个句子的起始位置和结束位置分别添加一个标志位 (BEGIN 和 END)，代表该句子的开始和结束。例如，句子 “the clouds are in the sky.” 将会转换成 ("BEGIN", "the", "clouds",





"are", "in", "the", "sky", ".", "END")。

大多数情况下，每一句话的长度都是不等的，有些句子由 10 个单词组成，有些句子则由 15 个单词组成，因此我们需要定义句子向量的长度。这里简单地将句子长度限定为 15 个单词，不足 15 个单词的句子需要在后面填充 PAD，多于 15 个单词的句子需要进行截断。例如句子 “the clouds are in the sky.” 转换成 ("BEGIN", "the", "clouds", "are", "in", "the", "sky", ".", "END", "PAD", "PAD", "PAD", "PAD", "PAD", "PAD")。

本例中的数据来源于 Julian McAuley 提供的 Amazon 网站中关于商品的问答情况，其数据的存储格式为 json 文件，数据存储方式如【代码清单 7-10】所示。本例中只使用问答数据中的 “question” 数据栏作为训练数据。

**【代码清单 7-10】Amazon product data 数据存储格式**

```
{
  "asin": "B000050B6Z",
  "questionType": "yes/no",
  "answerType": "Y",
  "answerTime": "Aug 8, 2014",
  "unixTime": 1407481200,
  "question": "Can you use this unit with GEL shaving cans?",
  "answer": "Yes. If the can fits in the."
}
```

在【代码清单 7-11】中，首先定义特殊的 Tokens 作为句子开头、句子结束、空白词汇和稀有词汇的标记，然后加载数据文档，并使用 question\_sent 列表存储前 1000 个 “question” 句子。

**【代码清单 7-11】加载 Amazon product data 数据**

```
import nltk
import json
import itertools
import numpy as np

# 特殊 Tokens 标注句子开头、句子结束、空白单词、稀有单词
start_token = "BEGIN"
end_token = "END"
pad_token = "PAD"
unknow_token = "UNKNOWN"

# 打开文本并读入句子
with open("qa_Appliances_quote.json") as json_file:
    json_data = json.load(json_file)
```



```
# 使用前 1000 个句子
question_sent = [x['question'].lower() for x in json_data][:1000]
```

读取数据后, 使用 NLTK 库把句子 Token 化, 并添加 START 在句子开头、END 在句子结束, 对没有满足 15 个单词的句子填充 PAD。最后是对词频进行统计, 使用 UNKNOW 替换稀有词汇, 并建立单词的双向索引字典 (index\_2\_word 和 word\_2\_index)。具体操作方式如【代码清单 7-12】所示。

**【代码清单 7-12】生成 Token 化句子并建立单词与索引之间的双向字典**

```
# 把句子 Token 化
tokenized_cent = [nltk.word_tokenize(x)[:13] for x in question_sent]

# 对每个句子添加句头标注、结束标注、空白标注
for i, cent in enumerate(tokenized_cent):
    cent.append(end_token)
    cent.insert(0, start_token)
    while(len(cent)<15):
        cent.append(pad_token)

# 取第一个句子进行显示
>>> tokenized_cent[0]
['BEGIN', 'does', 'this', 'work', 'with', 'the', 'insinkerator', 'evolution',
'septic', 'assist', '?', 'END', 'PAD', 'PAD', 'PAD']

# 统计词频, 得到 1000 个句子中出现过 1829 个不同的单词
word_freq = nltk.FreqDist(itertools.chain(*tokenized_cent))

# 建立双向索引 index_2_word、word_2_index
vocabulary_size = 1800 # 只使用常出现的 1800 个单词
vocab = word_freq.most_common(vocabulary_size) # 找到最常用的单词
index_2_word = dict([(i, w[0]) for i,w in enumerate(vocab)]) # 建立 index_2_word 索引
index_2_word[vocabulary_size] = unknow_token # 添加 unknow_token 到字典尾

word_2_index = dict([(index_2_word[i], i) for i, w in enumerate(index_2_word)])

# 用 unknow_token 替换未在词表中的词
for i, sent in enumerate(tokenized_cent):
    tokenized_cent[i] = [w if w in word_2_index else unknow_token for w in sent]
```



### 7.5.3 准备输入输出数据

上一节讲述了从文本到单词的预处理，本节讲述从单词向量到循环神经网络模型的输入。其输入是向量矩阵，而不是字符串数据，建立单词和索引之间的映射关系字典 `index_2_word` 和 `word_2_index` 是为了方便对循环神经网络模型的输入和输出进行单词索引。其操作方式如图 7-21 所示，用向量表示输入的单词后，还需要对单词向量进行转换才能送入循环神经网络模型的输入层。

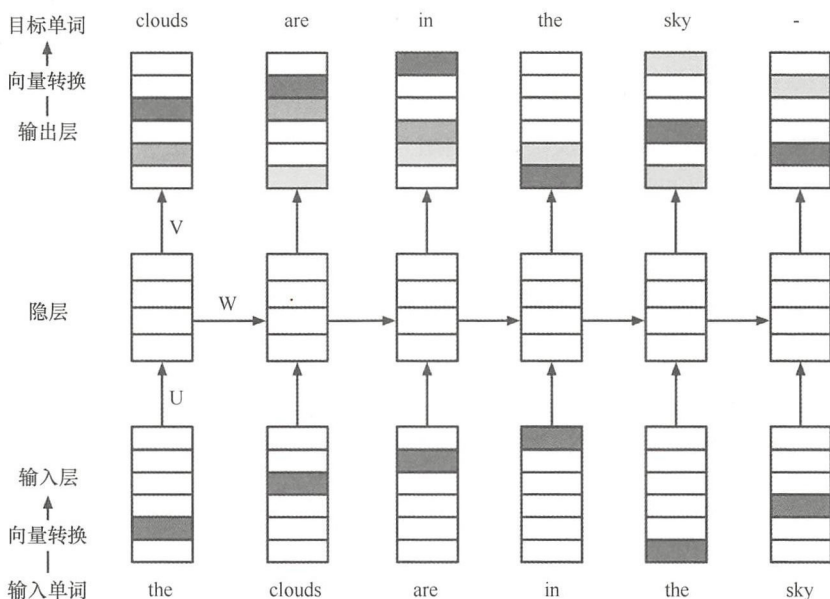


图 7-21 “the clouds are in the sky.” 的 RNN 网络模型

假设例句 “the clouds are in the sky.” 其单词索引的向量为  $[4, 808, 35, 21, 4, 809]$  (“the clouds are in the”), 预测目标是下一个单词，所以目标 Label 为  $[4, 808, 35, 21, 4, 809, 25]$  (“the clouds are in the sky”).

下面通过【代码清单 7-13】来看实际的训练数据和标签数据，训练集数据使用前 800 个句子，测试集使用后 200 个句子。

【代码清单 7-13】显示训练数据和标签数据

```
X_pre_train = np.asarray([[word_2_index[w] for w in sent[:-1]] for sent in
tokenized_cent[:800]])
Y_pre_train = np.asarray([[word_2_index[w] for w in sent[1:]] for sent in
tokenized_cent[:800]])
X_pre_text = np.asarray([[word_2_index[w] for w in sent[:-1]] for sent in
tokenized_cent[800:1000]])
```



```
Y_pre_test = np.asarray([[word_2_index[w] for w in sent[1:]] for sent in
tokenized_cent[800:1000]])
```

```
# train 训练数据
```

```
"BEGIN the clouds are in the sky . END PAD PAD PAD PAD PAD"
```

```
array([ 1, 4, 808, 35, 21, 4, 809, 25, 2, 0, 0, 0, 0, 0])
```

```
# label 预测数据
```

```
"the clouds are in the sky . END PAD PAD PAD PAD PAD"
```

```
array([ 4, 808, 35, 21, 4, 809, 25, 2, 0, 0, 0, 0, 0, 0])
```

循环神经网络模型的输入  $\mathbf{X}$  是一个单词序列, 每个输入  $x_i$  代表一个单词向量。但是在自然语言处理任务中不能直接使用单词的索引作为输入 (如输入为 [1, 4, 808, 35, 21, 4, 809, 25, 2]), 而应该经过编码转换为特定的向量形式作为循环神经网络模型的输入。在这里我们使用 one-hot 编码, 每个单词的向量长度为 vocabulary\_size+1, 其中 1 代表单词 UNKNOWN。例如单词 “sky” 的索引号为 31, one-hot 编码向量中第 31 个元素为 1, 其余元素为 0:

```
one-hot 单词向量: [ 0. 0. ... 0. 0. 1. 0. 0. ... 0. 0.]
```

```
对应词表字典位置: [ 0. 1. ... 29. 30. 31. 32. 33. ... 1799. 1800.]
```

接着通过 numpy 的 eye 函数对单词索引向量进行 one-hot 编码, 如【代码清单 7-14】所示。

#### 【代码清单 7-14】对单词索引向量进行 one-hot 编码

```
# one-hot 编码
```

```
X_train = np.eye(vocabulary_size+1)[X_pre_train]
```

```
X_test = np.eye(vocabulary_size+1)[X_pre_test]
```

```
Y_train = np.eye(vocabulary_size+1)[Y_pre_train]
```

```
Y_test = np.eye(vocabulary_size+1)[Y_pre_test]
```

```
# 800 个句子, 每个句子有 15 个单词, 每个单词用长度为 1800 的 one-hot 向量表示
```

```
>>> X_train.shape
```

```
(800, 14, 1801)
```

有了上述的知识, 我们就知道了本例对应循环神经网络模型的基本结构, 每一个时间序列的输入  $x_i$  代表一个 one-hot 单词向量, 因此输入  $\mathbf{X}$  为一个  $M \times N$  矩阵,  $M$  表示句子的单词数,  $N$  表示 one-hot 向量的长度。假设每个句子有 15 个单词, one-hot 向量长度为词表字典长 1800, 因此输入  $\mathbf{X}$  为  $15 \times 1800$  的矩阵。

同理, 经过循环神经网络模型后输出  $\mathbf{H}$  也是一个矩阵, 其中时序  $h_i$  表示该单词在 one-hot 向量的概率, 因此本例中的网络模型最终如图 7-21 所示。





### 自然语言处理为什么不使用索引作为输入？

网络的输出往往是一个概率分布，然后选取概率最高的值作为最终输出，从而要求输入的数据也是以概率分布的形式呈现，而不是以单词索引向量呈现。另外一个原因是对齐输入数据格式，方便矩阵运算。

根据单词出现的先后顺序，可以将每一个单词视为对应的一个时间序列。每一个时间序列需要计算一次隐层状态  $s_t$  和输出  $h_t$  的值。本例中加入单词 PAD 使得句子对齐为 15 个单词向量，因此一次前馈实际上进行了 15 次向前传播计算。

【代码清单 7-15】对训练数据的第 100 个句子进行预测得到输出矩阵，计算预测函数得到词表索引数组。最后通过索引得到预测句子 'machine also live mini-fridge this dispense fix easily unable travel travel PAD PAD PAD'。由于循环神经网络模型的权重参数  $U$ 、 $W$ 、 $V$  是随机初始化的，因此现在的预测值是随机的。

#### 【代码清单 7-15】循环神经网络向前传播后输出预测句子

```
>>> rnn_model = SimpleRNN(vocabulary_size + 1)
>>> y_predict = rnn_model.predict(X_train[100]) # 模型预测

array([1762, 749, 1340, 1516, 1517, 1129, 1451, 366, 1557, 382, 623, 133, 1782, 1477])

>>> y_predict_sent = [index_2_word[i] for i in y_predict] # 根据单词索引表找到对应单词
>>> print(' '.join(y_predict_sent)) # 把单词数组连接成字符串

'machine also live mini-fridge this dispense fix easily unable travel PAD PAD PAD'
```

## 7.5.4 实现简单的循环神经网络模型

本例的目标是通过循环神经网络模型来预测文本中下一个可能出现的单词，获得出现目标单词的概率，结果如图 7-22 所示。

在【代码清单 7-16】中，首先定义 Keras 中的序列模型 Sequential()，接着定义隐层网络使用哪种类型的隐层。该例子中采用 Keras 自带的 SimpleRNN 网络层，输入是 X\_train 的后两个维度数据 [14, 1801]，其中 14 为序列数，也称为时间步  $t$ ；1801 是每一时间步  $t$  的特征值，本例中使用单词的 one-hot 向量特征。循环神经网络模型中的隐层节点数设置为 256 个。

输出使用了 TimeDistributed 层，其中每一个 TimeDistributed 中的输出为 Y\_train.shape[2]，也就是 1801 单词的 one-hot 向量特征，输出的激活函数选择了 Softmax 函数。



## 【代码清单 7-16】Keras 中循环神经网络模型

```
import numpy as np
from keras.models import Sequential
from keras.layers import SimpleRNN as RNN
from keras.layers import Dense, Activation, TimeDistributed

# 建立循环神经网络模型
print('Build Simple RNN model...')

rnn_model = Sequential()
rnn_model.add(RNN (256, input_shape=X_train.shape[1:], return_sequences=True,
name='RNN'))
rnn_model.add(TimeDistributed(Dense(Y_train.shape[2], activation='softmax'),
name='softmax'))

rnn_model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
print(rnn_model.summary())
```

Build Simple RNN model...

Layer (type)	Output Shape	Param #
RNN (SimpleRNN)	(None, 14, 256)	526848
softmax (TimeDistributed)	(None, 14, 1801)	462857
Total params: 989,705.0		
Trainable params: 989,705		
Non-trainable params: 0.0		
None		

图 7-22 本例中循环神经网络的基本信息，隐层中使用了 14 个时间序列，隐层神经元为 256 个，共使用了 526848 个参数；输出层同样为 14 个时间序列，输出特征为 1801 的向量，使用了 462857 个参数。从信息中可以大约估算出该网络模型的性能和瓶颈

在训练阶段，使用 `fit` 对定义的循环神经网络模型进行训练，训练迭代的次数为 30 次 (`nb_epoch=30`)，每次训练 64 条数据 (`batch_size=64`)，如【代码清单 7-17】所示。该例执行内核使用了 TensorFlow，在作者的计算机上执行效率约为 5s 迭代一次，最终的效果如图 7-23 所示。

## 【代码清单 7-17】RNN 模型训练

```
>>> rnn_model.fit(X_train, Y_train, nb_epoch=30, batch_size=64)
```

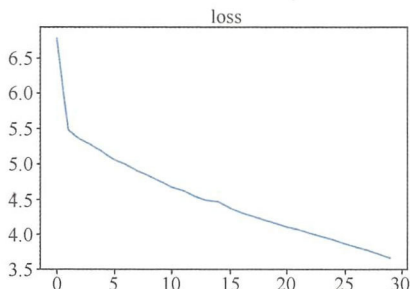
Epoch 1/30



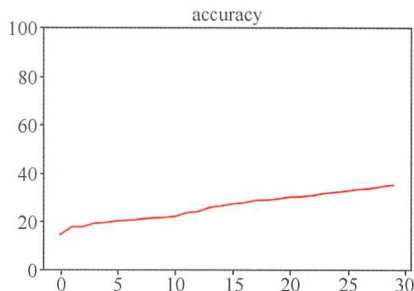
```

800/800 [=====] - 5s - loss: 6.7772 - acc: 0.1446
Epoch 2/30
800/800 [=====] - 4s - loss: 5.4762 - acc: 0.1768
Epoch 3/30
800/800 [=====] - 5s - loss: 5.3508 - acc: 0.1767
...
Epoch 28/30
800/800 [=====] - 5s - loss: 3.7729 - acc: 0.3366
Epoch 29/30
800/800 [=====] - 5s - loss: 3.7153 - acc: 0.3454
Epoch 30/30
800/800 [=====] - 5s - loss: 3.6581 - acc: 0.3510

```



(a) RNN网络模型迭代损失输出值



(b) RNN网络模型迭代预测准确率

图 7-23 循环神经网络模型训练结果

训练完循环神经网络模型后，我们需要输出显示该神经网络模型，查看训练后的内容。如【代码清单 7-18】所示，拿到输出层的数据之后需要使用 `index_2_word` 索引到单词字典，对输出的单词向量进行可读性输出。

#### 【代码清单 7-18】对循环神经网络训练结果输出显示

```

# 应该输出的向量
>>> input_yy = np.argmax(Y_train[0], axis=1)
[ 4 808 35 21 4 809 25 2 0 0 0 0 0 0]

# 模型预测输出的向量
>>> predict_yy = np.argmax(train_predict[0], axis=1)
[ 8 5 15 4 6 3 3 2 0 0 0 0 0 0]

# 应该输出的向量转换为实际句子
>>> [index_2_word[i] for i in input_yy]
['the', 'clouds', 'are', 'in', 'the', 'sky', '.', 'END', 'PAD', 'PAD', 'PAD',
'PAD', 'PAD', 'PAD']

```



```
# 模型预测输出的向量转换为实际句子
>>> [index_2_word[i] for i in predict_yy]
['does', 'this', 'fit', 'the', 'a', '?', '?', 'END', 'PAD', 'PAD', 'PAD',
 'PAD', 'PAD', 'PAD']
```

从网络的训练效果图 7-23 (a) 中可以看出, 其实网络损失值还没有完全停止下降, 循环神经网络模型已经停止训练, 因此想要继续减少损失, 可以把迭代次数继续提高; (b) 图为模型的预测精度, 随着损失越来越少, 精度也是一直在上升, 可是精度上升的幅度不大 (14% ~ 35%), 预测结果比随机预测精度 50% 还要低。

若要进一步提高循环神经网络模型的预测精度, 有以下 4 个方法:

- 增加模型训练迭代时间;
- 增加训练数据, 本例中训练的数据集只有 800 条, 读者可以尝试将训练集的数据增加到 8 万条, 看预测准确率能否超过 70%;
- 增加隐层的节点数, 产生更复杂的模拟函数;
- 把记忆单元换为 LSTM, 能够处理时序上长距离依赖的问题, 增强循环神经网络模型的记忆能力。

## 7.6 本章小结

本章主要概述了基本的循环神经网络以及其核心算法。从构建一个循环神经网络模型开始, 到使用 BPTT 时间反向传播算法, 以及在语言模型上应用循环神经网络模型。然而, 循环神经网络并没有就此结束, 基本的循环神经网络存在很严重的梯度爆炸和梯度消失问题, 并不能真正处理时序上长距离依赖的数据。事实上, 真正得到广泛应用的是循环神经网络模型的一个变体——长短期记忆网络 (LSTM)。LSTM 可以很好地解决序列数据距离依赖这一问题, 我们将在下一章中详细介绍。

- 序列数据: 有先后关联关系或者时间关系的信号数据, 都可以认为是序列数据。常见的序列数据有文字、语音等。
- 循环神经网络: 增加了隐层记忆单元, 使得序列中一个元素都是相关的; 当前的输出依赖于上一时间步的输出, 使得循环神经网络能够对序列数据进行建模。解决了前馈式神经网络不能够对序列信号建模和缺乏数据反馈的缺点。
- 循环神经网络的基本结构: 由输入向量  $x$ 、隐层状态  $s$ 、输出向量  $h$  组成, 其中输入层与隐层通过权重  $U$  连接, 隐层状态之间通过权重  $W$  连接, 隐层到输出层则通过权重  $V$  连接。





- 记忆单元：循环神经网络结构中的隐层状态作为核心层，具有对序列信号的记忆功能，因此也被称为记忆单元。其中，隐层状态  $s_t$  是根据上一个时刻  $t-1$  的隐层状态  $s_{t-1}$  和当前时刻  $t$  的输入  $x_t$  计算得到的。
- 循环神经网络的向前传播：输入层接收序列数据输入，根据序列先后顺序进行向前传播，每一时间步需要计算一次隐层状态  $s_t$  和输出  $h_t$  的值，然后把隐层状态  $s_t$  依次传给下一个时间步传播计算。
- 循环神经网络的总损失  $L$  为每一时间步  $t$  的损失值  $L_t$  的和。
- BPTT 算法与 BP 算法类似，增加了根据时间进行反向传递的功能，用于求解循环神经网络模型中 3 个权重参数  $U$ 、 $W$ 、 $V$  的导数。
- 因为梯度爆炸或者梯度消失的问题，循环神经网络模型很难学习到长期依赖的序列数据。虽然梯度问题不可避免，但是我们可以有多种方法弱化梯度计算所引起的问题。

## 引用/参考

- [1] Le Q V, Jaitly N, Hinton G E. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units[J]. Computer Science, 2015.
- [2] Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks[C]// International Conference on International Conference on Machine Learning. JMLR.org, 2013:III-1310.
- [3] Graves A, Jaitly N. Towards end-to-end speech recognition with recurrent neural networks[C]// International Conference on Machine Learning. 2014:1764-1772.
- [4] Wang W, Cui Z, Yan Y, et al. Recurrent Face Aging[C]// Computer Vision and Pattern Recognition. IEEE, 2016:2378-2386.
- [5] Schank R C, Goldman N, Riager C J, et al. Margie memory, analysis, response generation, and inference on English[C]// International Joint Conference on Artificial Intelligence. Morgan Kaufmann Publishers Inc. 1973:255-261.
- [6] Mcauley J, Yang A. Addressing Complex and Subjective Product-Related Queries with Customer Reviews[C]// International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, 2016:625-635.

---

# 第 8 章

---

## 循环神经网络 进阶序列长期 记忆

本章主要内容：

- 长短期记忆网络（LSTM）
- 门控循环单元（GRU）

循环神经网络（Recurrent Neural Network, RNN）因其循环递归地处理历史数据，并对历史记忆进行建模的特殊性能，而特别适用于处理时间、空间序列上有强关联的信息。然而仅凭一个简单的循环神经网络模型并不能完美地解决复杂的序列数据，序列数据的时间复杂度不断增加，会给网络带来一定的影响。

为了解决序列数据的长期依赖问题，部分学者尝试对长期依赖的序列数据进行转换，减短序列数据长度，使得长依赖的数据变成短依赖的数据。可是这不能从根本上解决问题，长依赖数据转换成短依赖数据必然意味着部分数据信息的丢失，而造成原数据失真。为了解决长期依赖问题，（Sepp Hochreiter et al., 1997）等学者提出了“记忆单元（Memory Cell）”的概念，用于存储记录长依赖的数据信息。经过了数十年的发展，循环神经网络模型出现了众多变体，各种循环神经变体网络模型的性能也各有千秋。

本章会首先介绍长期依赖的问题，然后引入近年来非常热门的长短期记忆网络（LSTM），接着介绍对 LSTM 精简后的门控循环单元（GRU）。在示例中，将会介绍使用基于 LSTM 的网络模型来实现基于神经网络的机器翻译系统，最后我们将会利用 GRU 等新兴的网络架构实现自动问答机器人系统。

在本章中，我们可以从循环神经的不同网络架构上吸取非常特别的网络设计思想和先进理念。序列信息无处不在，翻开下一页，让我们继续拥抱序列数据！

## 8.1 长期依赖问题

循环神经网络模型的核心思想是根据不同时间的序列信息来更新网络中的参数，使得循环神经网络模型学习到序列数据之间的关联信息，进而推测未来将要出现的序列信息。如根据说过的话来推测接下来要说的词语，或者根据历史股票收盘交易记录数据预测明天的股票收盘数值。

如果循环神经网络模型能够完美解决序列数据问题，将会对人工智能领域具有巨大的推动作用。循环神经网络模型提出的概念确实很先进，在理论上可以处理序列数据，但是随着其网络模型的发展，梯度消失和梯度爆炸所引起的问题也随之而来。

我们将普通循环神经网络模型进行展开，可以将其视为一个所有层共享权值的深度前馈神经网络。虽然循环神经网络模型的目的是学习序列信息中的关联关系，但根据理论分析和实际工程验证的结果，普通循环神经网络模型很难学习和保存长期序列信息。下面来看看短期记忆和长期记忆。

## 1. 短期记忆

假设现有一个语言模型，基于之前出现过的单词来预测下一个将会出现的单词，例如预测句子“The clouds are in the\_\_\_\_\_”中下划线处可能出现的单词。根据上一章的内容，模型应该预测单词“sky”的概率最高。在序列信息短、预测单词间的间隔短（例子中为1）的语境中，该类型数据称为短期序列，循环神经网络模型处理短期序列的过程称为“短期记忆”。图8-1所示为使用循环神经网络模型预测短期序列数据，例中序列的总长度为5个时间步（ $T=5$ ），输入的序列信息为 $x_0$ 和 $x_1$ ，需要对时间步为4的输出进行预测，预测序列间隔为3。

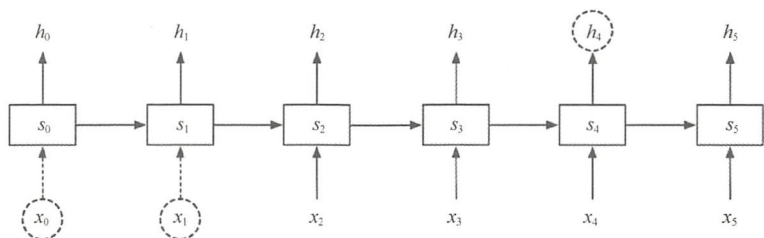


图 8-1 使用循环神经网络模型预测短期序列数据

循环神经网络模型可以很好地学习短期序列信息，并且能够轻易地达到70%以上的预测精度。可当序列数据信息很长、预测间隔大时，循环神经网络模型的预测效果又如何呢？这里我们引入了长期记忆问题。

## 2. 长期记忆

例如在一个更加复杂的语境中，“她是一个经常出现在我梦中，美丽动人、温柔贤淑、端庄大雅的女人。”我们尝试根据前面出现的词语序列“她是”去预测最后出现“女人”这个词语的概率。如图8-2所示，假设“她”为 $x_0$ ，“是”为 $x_1$ ，我们已经知道序列 $x_0$ 和序列 $x_1$ 的位置信息，需要预测 $H_{t+1}$ （“女人”）的信息。在该情况下，相关信息与当前预测位置之间的间隔非常大，由于信息随着时序传播会逐渐引起梯度消失或者梯度爆炸问题，所以循环神经网络模型难以处理长期记忆的任务。

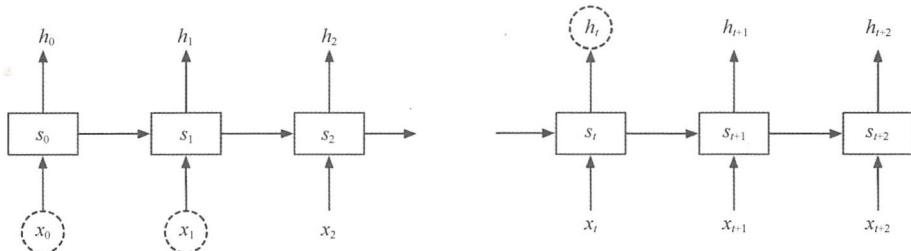


图 8-2 使用循环神经网络模型预测长期序列数据



根据上一章的知识我们知道，随着预测序列间隔增大，循环神经网络模型就会引起 BPTT 时间反向传播算法中的梯度消失和梯度爆炸问题。其中一个有趣的概念叫作梯度消失（Vanishing Gradient），循环神经网络训练的关键是 BPTT 算法，梯度信息在时间序列上反向传播的过程中不断地衰减其数值。数据信息传递效果的好坏取决于传递时数值衰减速度的快慢，理论上循环神经网络模型可以处理很长的信息，但是由于梯度消失问题，往往事与愿违。

如图 8-3 所示，我们将循环神经网络模型展开，序列从左到右，图中网络模型的节点阴影代表对各自的时间序列  $t$  的输入数据敏感度（阴影越深，敏感度越高）。随着时间序列的传递，阴影颜色越来越淡，也就是对原来输入的数据敏感度越少，最后循环神经网络模型已经忘记了开始序列的信息。因此，普通的循环神经网络模型只能处理简单的短期记忆问题，而不能有效地处理长序列数据问题。

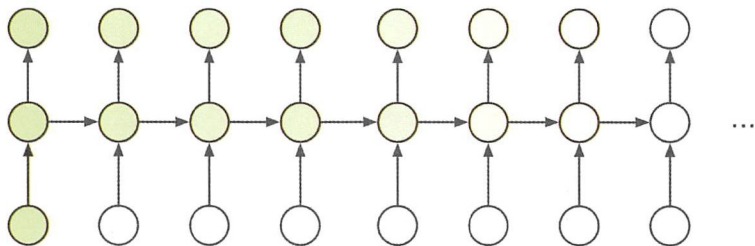


图 8-3 循环神经中梯度消失问题

为了解决长期依赖（Long-term Dependencies）问题，研究学者提出了多种循环神经网络的变体。其中，（Sepp Hochreiter et al., 1997）提出长短期记忆网络（Long Short-Term Memory Networks, LSTM），目的是保存长期序列信息，减少随着时间传播而衰减的信息，需要给循环神经网络增加记忆功能，使其能长期记住输入信息的状态。经过了十几年的发展，LSTM 被证明比传统的循环神经网络模型在处理长期依赖问题上更加有效，LSTM 自然而然地成为了循环神经网络中最为经典的网络模型。

（Alex Graves et al.）在 2012 年对 LSTM 进行了改进，使得 LSTM 网络得到了广泛的应用。另外由（Cho, et al.）在 2014 年提出的门控循环单元（GRU），最终的模型比标准的 LSTM 模型要简单，也成为了近年来非常流行的循环神经网络模型之一。如图 8-4 所示，从左到右分别为普通的循环神经网络模型、LSTM 网络模型、GRU 网络模型的示例图，相信读者已经对普通循环神经网络模型有一定的理解，下面我们来深入探讨 LSTM 网络模型和 GRU 网络模型。

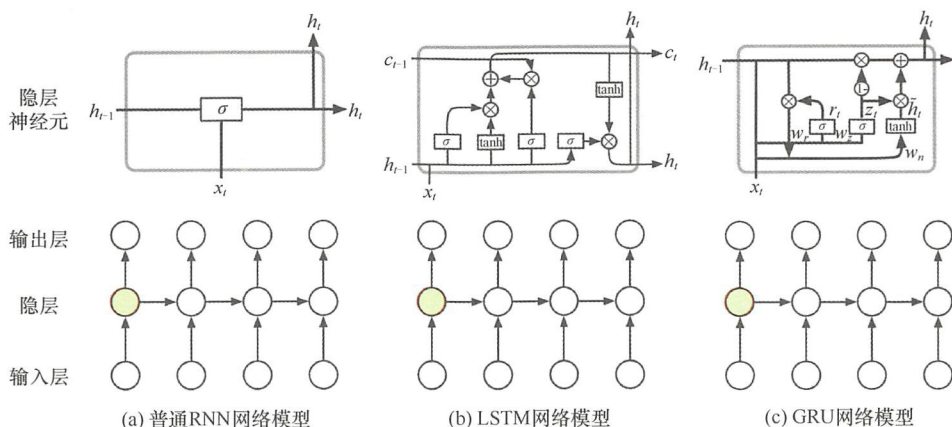


图 8-4 (a) 为循环神经网络模型, (b) 为 LSTM 网络模型, (c) 为 GRU 网络模型。在循环神经网络模型中的隐层神经元为简单的激活函数  $\sigma$ , LSTM 使用记忆单元 (Memory Cells) 代替了循环神经网络的隐层神经元, GRU 的记忆单元则是对 LSTM 进行精简

## 8.2 长短期记忆网络

长短期记忆网络 (Long Short-Term Memory, LSTM), 顾名思义, 是增加了记忆功能的循环神经网络模型。LSTM 与循环神经网络在时序上的传播方式并没有本质上的差异, 区别在于它们使用不同的方式去计算隐层神经元状态。LSTM 中的记忆功能被称为 “Cells”, 这些 Cells 在神经元内部决定是否应该写入或删除对信息的记忆, 并且可以将之前的状态、现在的记忆和当前输入的信息结合在一起, 对长期信息进行记录。

### 8.2.1 LSTM网络结构

上面提到 LSTM 中的记忆功能被称为 “Cells”, 这里统一叫作 “记忆单元 (Memory Cells)”。记忆单元如图 8-5 所示, 主要由 4 个元素组成: 输入门 (Input Gate)、遗忘门 (Forget Gate)、输出门 (Output Gate) 和自循环连接神经元 (Self-recurrent Connection)。其中, 门 (Gate) 把输出范围控制为在 0 到 1 之间的数值, 负责描述每个部分有多少量可以通过。数值 0 代表 “不允许任何量通过”, 数值 1 代表 “允许任意量通过”, LSTM 通过控制 3 个门的输出值来保护和控制记忆单元状态。

下面进一步对图 8-5 所示的记忆单元进行分析。自循环连接神经元的作用是为了让记忆单元随着时间的推移，序列信息的输出仍然保持独立，不受输入和输出环境的影响；输入门是决定哪些新输入的信息允许被更新，或者被保存到记忆单元中；与输入门相对的是输出门，用于决定记忆单元中哪些信息允许被输出；输出门则用于控制记忆单元是否记住或者丢弃之前的状态。

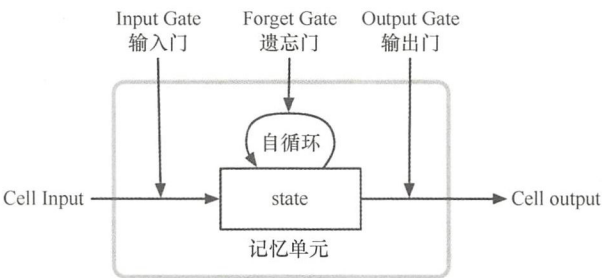


图 8-5 记忆单元组织图。由 4 个元素组成：输入门（Input Gate）、遗忘门（Forget Gate）、输出门（Output Gate）和自循环连接神经元（Self-recurrent Connection）

## 8.2.2 LSTM记忆单元

下面以时间步  $t$  为例，介绍 LSTM 中的记忆单元存储序列信息的过程，并对图 8-5 进行展开，得到如图 8-6 所示的 LSTM 记忆单元详细组织图，图中用到的数学符号如下。

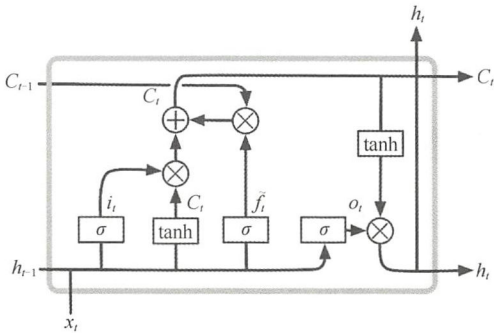


图 8-6 LSTM 记忆单元详细组织图

- $x_t$ ：在时间步  $t$  记忆单元的输入。
- $i_t$ ：输入门的激活值。

- $f_t$ : 遗忘门的激活值。
- $o_t$ : 输出门的激活值。
- $h_t, h_{t-1}$ : 在时间步  $t$  和时间步  $t-1$  记忆单元的输出。
- $C_t, C_{t-1}$ : 时间步  $t$  和时间步  $t-1$  记忆单元的状态。
- $\tilde{C}_t$ : 记忆单元的候选状态。
- $W_i, U_i, W_c, U_c, W_f, U_f, W_o, U_o$ : 分别记忆单元中对应门的权重向量（图中并没有标出）。
- $b_i, b_c, b_f, b_o$ : 分别为记忆单元中对应门的偏置（图中并没有标出）。

### 1. 输入门

输入门决定哪些新输入的信息允许被更新，或者被保存到记忆单元中。为了确定什么样的新信息可以被保存在记忆单元中，如图 8-7 中红色部分所示，需要计算输入门的激活值  $i_t$  和时间步  $t$  记忆单元的状态候选值  $\tilde{C}_t$ ：

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \quad (8-1)$$

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \quad (8-2)$$

在式 (8-1) 中， $\sigma$  为激活函数， $W_i$  为在输入门输入控制时间步  $t$  的输入序列数据的权重向量， $U_i$  为在输入门输入控制时间步  $t-1$  输入状态值的权重向量， $b_i$  为输入门输入控制的偏置。在式 (8-2) 中， $W_c$  为在输入门状态候选在时间步  $t$  的输入序列数据的权重向量， $U_c$  为在输入门状态候选时间步  $t-1$  输入状态值的权重向量， $b_c$  为输入门状态候选的偏置。

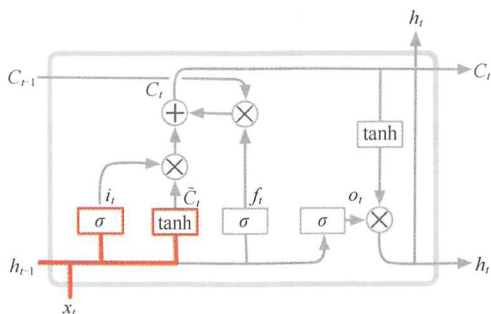


图 8-7 输入门和状态候选值的计算过程的数据流图

虽然式 (8-1) 和式 (8-2) 看上去很复杂，实际上两公式相类似，区别在于其权重向量，一个是针对激活值  $i_t$ ，另外一个是对记忆单元的状态候选值  $\tilde{C}_t$ 。其中，在计算输入门的控制输出值和状态候选值时，输入都为当前时间步  $t$  的数据信息  $x_t$  和上一时间步  $t-1$  的状态输出  $h_{t-1}$ 。



## 2. 输出门

用于控制记忆单元是否记住或者丢弃之前的状态。输出门的作用与名字相符，决定从记忆单元中丢弃哪些信息，计算时读取当前时间步  $t$  的输入数据信息  $x_t$  和上一时间步  $t-1$  的状态输出  $h_{t-1}$ ，输出  $0 \sim 1$  之间的数值作为上一次记忆单元的状态。

接下来需要计算在时间步  $t$  记忆单元处的输出门的激活值  $f_t$  和新的状态值  $C_t$ 。

在输入门中，我们得到了输入激活值  $i_t$  和记忆单元的状态候选值  $\tilde{C}_t$ ，如图 8-8 中红色部分所示，需要计算输出门的激活值  $f_t$  和当前时间步  $t$  的新状态值  $C_t$ ：

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (8-3)$$

$$C_t = i_t \times \tilde{C}_t + f_t \times C_{t-1} \quad (8-4)$$

在式 (8-3) 中， $\sigma$  为激活函数，其原理与式 (8-1) 和式 (8-2) 相同。在式 (8-4) 中，当前时间步的新状态值  $C_t$  为上一时间步的状态值  $C_{t-1}$  与输出门的激活值  $f_t$  相乘，作用为决定丢弃旧状态中的信息量；输入门输出控制值  $i_t$  与候选状态值  $\tilde{C}_t$  相乘，用于控制新状态的变化；最后把两者相加作为新的状态值。

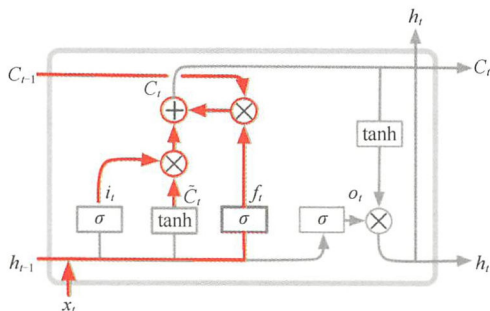


图 8-8 输出门和新状态值的计算过程的数据流图。输出门决定丢弃的信息，其结合旧的状态值和输入门的激活值，得到新状态值

## 3. 输出门

决定记忆单元中哪些信息允许被输出。输出门的作用与输入门对称。如图 8-9 中红色部分所示，需要计算时间步  $t$  记忆单元中输出门的输出激活值  $o_t$  和记忆单元的输出值  $h_t$ ：

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad (8-5)$$

$$h_t = o_t \times \tanh(C_t) \quad (8-6)$$

在式 (8-5) 中， $\sigma$  为激活函数，其原理与式 (8-1) 和式 (8-2) 相同。在式 (8-6) 中，输出门的输出激活值与当前时间步  $t$  新状态值的 Tanh 函数相乘，最终得到输出

状态  $h_t$ 。

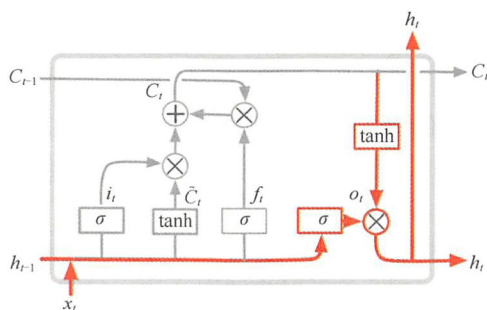


图 8-9 输出门和记忆单元输出值的计算过程的数据流图

以上是 LSTM 的基本记忆单元计算方式，然后并不是所有的 LSTM 网络模型都如上述形式。实际上，当一个性能较好的网络结构被提出来之后，可能会出现众多研究学者对该基本模型进行改进和升级。比较流行的 LSTM 网络是通过增加孔连接 (Peephole Connection) (如图 8-10 红色部分所示)，使得输入门、输出门、输出门都接受上一时间步的记忆单元状态作为输入，(Gers et al., 2000) 的实验表明，增加孔连接能够提高 LSTM 的学习精度和记忆能力，最终 3 个门的激活值公式为：

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + \underline{V_i C_{t-1}} + b_i) \quad (8-7)$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + \underline{V_f C_{t-1}} + b_f) \quad (8-8)$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + \underline{V_o C_{t-1}} + b_o) \quad (8-9)$$

其中  $C_{t-1}$  为上一时间步记忆单元的状态值，图 8-10 中的 3 个箭头从左到右分别为  $V_i$ 、 $V_f$ 、 $V_o$ ，代表对于 3 个门控的不同权重值。

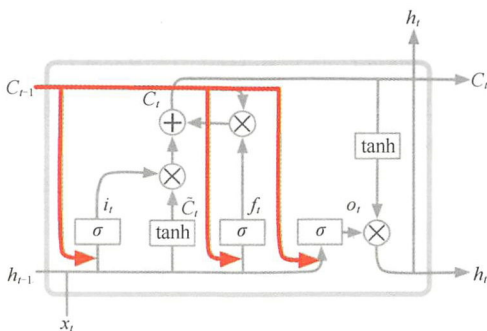


图 8-10 增加孔连接 (Peephole Connection) 计算过程的数据流图

## 8.2.3 LSTM记忆方式

门的作用是允许 LSTM 的记忆单元长时间存储和访问序列信息，从而减少梯度消失问题。例如，输入门保持关闭（即激活值接近 0），则新的输入不会进入网络，网络中的记忆单元会一直保持开始的激活状态。通过对输入门的开关控制，可以控制循环神经网络模型什么时候接受新的数据、什么时候拒绝新的数据进入，于是梯度信息就随着时间的传递而被保留下来。

如图 8-11 所示，在中间隐层节点的单个记忆单元中，左侧为输出门、下侧为输入门、上侧为输出门。当门完全打开时为圆圈“o”，门完全关闭时为横杠“=”。在第 2 个时间步  $t$  中，输入门和输出门都为关闭状态，于是记忆单元把之前的状态传递到第 3 个时间步  $t+1$ ，操作中记忆单元就把第 1 个状态记录下来。依次类推，记忆单元就能够对时间步  $t$  的状态进行存储，减少梯度消失问题。

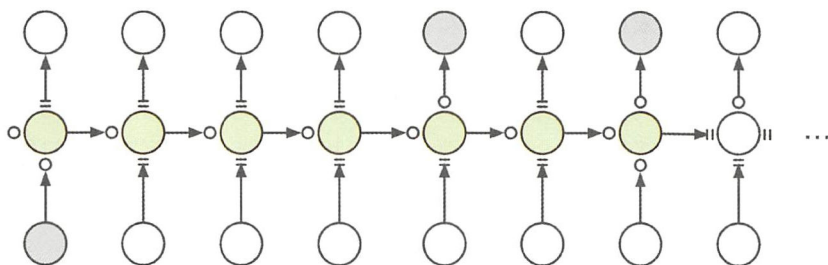


图 8-11 LSTM 保存梯度信息示例图

## 8.3 门控循环单元

门控循环单元（Gated Recurrent Unit, GRU）将 LSTM 模型的门控信号减少到了 2 个，分别称为更新门（Update Gate）和复位门（Reset Gate）。也可以说，GRU 是对 LSTM 的精简，有了对 LSTM 计算过程的数据流图认识后，门控循环单元计算过程的数据流图（如图 8-12 所示）就显得相对简单了。

下面以时间步  $t$  为例，介绍 LSTM 中的记忆单元进行序列信息存储的过程，如图 8-12 所示为 GRU 的详细组织图，图中用到的数学符号如下。

- $t_t$ ：在时间步  $t$  记忆单元的输入。
- $r_t$ ：复位门的激活值。
- $z_t$ ：更新门的激活值。

- $\tilde{h}_t$ : 记忆单元中的更新输出候选值。
- $h_t, t_{t-1}$ : 在时间步  $t$  和时间步  $t-1$  记忆单元的输出。
- $W_r, W_z, W_h$ : 分别记忆单元中对应门控的权重向量。

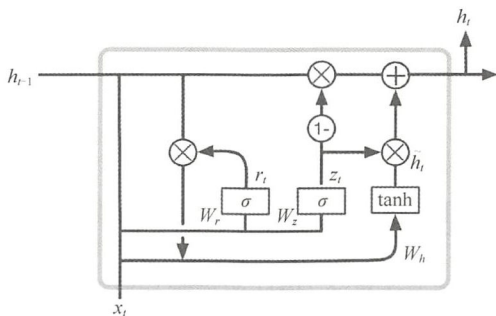


图 8-12 GRU 计算过程的数据流程图

## 8.3.1 GRU记忆单元

在 GRU 中只有两个门，分别是复位门和更新门。其中，复位门决定如何将新的输入数据和旧的记忆信息相结合，更新门则决定保留多少前面的记忆量。如果网络中复位门全为 1，更新门全为 0，那么 GRU 就相当于普通的 RNN 网络。

### 1. 复位门

决定如何将新的输入数据信息与旧的记忆信息相结合。需要计算在时间步  $t$  记忆单元中的重置激活值  $r_t$ ：

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \quad (8-10)$$

$r_t$  用于控制前一时间步隐层单元  $h_{t-1}$  对当前输入数据  $x_t$  的影响。如果  $h_{t-1}$  对  $x_t$  不重要，即从当前输入数据  $x_t$  开始表述了新的意思，并且与上文无关，那么  $r_t$  开关可以保持打开状态，使得  $h_{t-1}$  对  $x_t$  不产生影响。

### 2. 更新门

用于决定是否使用当前时间步  $t$  的输入信息  $x_t$  对网络产生影响。其更新激活值  $z_t$  为：

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \quad (8-11)$$

$z_t$  和  $r_t$  的计算公式类似。其中，权重参数  $W$  针对时间步  $t$  的输入数据，权重参数  $U$  针对循环记忆单元上一时间步的记忆信息  $h_{t-1}$ ， $z_t$  用于决定是否忽略当前时间步的输入数据  $x_t$ 。类似 LSTM 中的输入门  $i_t$ ， $z_t$  可以判断当前输入数据  $x_t$  对整体序列信息



的表达是否重要。当  $z_t$  开关打开时，我们将会将忽略当前输入数据  $x_t$ ，同时让  $h_{t-1}$  与  $h_t$  相连通，使得梯度能够得到有效的反向传播。和 LSTM 相同，这种短路机制有效地缓解了梯度消失现象。

最后需要计算在时间步  $t$  记忆单元中的更新输出候选值  $\tilde{h}_t$  和记忆单元信息输出值  $h_t$ ：

$$\tilde{h}_t = \tanh(W_h x_t + U_h r_t \cdot h_{t-1}) \quad (8-12)$$

$$h_t = (1 - z_t) h_{t-1} + z_t \tilde{h}_t \quad (8-13)$$

## 8.3.2 GRU实现

【代码清单 8-1】为门控循环单元 GRU 的前馈代码和梯度计算代码。前馈代码函数为 GRU\_forward()，函数中所使用到的参数有 W、U、b、c。其中，W 和 U 为记忆单元里的权重参数，b 和 c 为偏置。z\_t、r\_t 分别为更新门和复位门的计算公式。

代码中定义输入的序列数据长度为 word\_dim=10，隐层单元数只有 1 个。

【代码清单 8-1】GRU 的向前传播算法

```
# 定义参数
word_dim = 10
hidden_dim = 1
c = np.zeros(word_dim)
E = np.random.uniform(-np.sqrt(1./word_dim), np.sqrt(1./word_dim), (hidden_dim, word_dim))

def GRU_forward(x_t, h_t_prev):
    # 在 simple RNN 中前馈计算使用的是
    # h_t = np.tanh(U.dot(x_t) + W.dot(h_t_prev))

    # 获取时间步 t 的向量
    x_t_ = E[:, x_t]

    # GRU
    z_t = sigmoid(U[0].dot(x_t_) + W[0].dot(h_t_prev) + b[0]) # 更新门
    r_t = sigmoid(U[1].dot(x_t_) + W[1].dot(h_t_prev) + b[1]) # 复位门
    c_t = np.tanh(U[2].dot(x_t_) + W[2].dot(h_t_prev * r_t) + b[2])
    h_t = (1 - z_t) * c_t + z_t * h_t_prev

    # 最后输出概率
    x_t = softmax(V.dot(h_t) + c)[0]
    return [x_t, s_t1]
```

前馈计算函数较为简单，只是对图 8-12 中 GRU 计算过程的数据流进行实现。由于梯度的计算较为复杂，这里就不再根据求导的链式法则进行计算推导。对于导数的计算，【代码清单 8-2】使用了 Keras 后端的 `grad()` 函数，Keras 调用 TensorFlow 的梯度计算函数，输入的是 `loss` 损失函数的公式和被求导的参数。

【代码清单 8-2】GRU 的参数梯度求导

```
def GRU_gradient(y_pred, y):
    # 向后传播
    loss = categorical_crossentropy(y_pred, y)

    # 计算权重参数梯度
    dE = keras.grad(loss, E)
    dU = keras.grad(loss, U)
    dW = keras.grad(loss, W)
    db = keras.grad(loss, b)
    dV = keras.grad(loss, V)
    dc = keras.grad(loss, c)
```

### 8.3.3 GRU与LSTM比较

尽管循环神经网络、LSTM 和 GRU 的网络结构差别很大，但是它们的基本计算单元是一致的，都是对前一时间步隐层单元  $h_{t-1}$  和当前输入数据  $x_t$  做一个线性映射加激活函数。区别在于如何设计额外的门控机制控制梯度信息传播的方式，以缓解梯度消失现象。

GRU 和 LSTM 两个模型的出现都是为了解决序列长期依赖引起的梯度消失问题，GRU 与 LSTM 的区别如下：

- GRU 单元中只有 2 个门，而 LSTM 单元中有 3 个门；
- GRU 没有中间状态值  $C_t$ ，LSTM 则把中间状态值  $C_t$  传递给下一隐层状态；
- LSTM 在计算输出时使用了 Tanh 非线性函数，而 GRU 未使用。

上面虽然说了几点 GRU 与 LSTM 的架构设计区别，但还未涉及应用上的区别。从论文发表的时间来看，GRU 在 2014 年被提出，LSTM 的形态则是从 1997 年被提出，到近年来才固定下来，时期不同，应该说针对的应用数据也不同。测试结果表明，GRU 网络中的参数比 LSTM 少很多，因此训练的速度更快，收敛时间短，不需要太多训练的数据就可以得到很不错的效果。如果想进一步提高精度，哪怕是一个百分点，那么可以准备更多的序列数据或者使用网络参数更加负责的 LSTM。在深度学习中，我们需要根据所拥有的实际数据和训练目标来决定使用什么样的网络架构。

## 8.4 示例1: 神奇的机器翻译

根据维基百科对机器翻译的定义:

*Machine translation, sometimes referred to by the abbreviation is a sub-field of computational linguistics that investigates the use of software to translate text or speech from one language to another.*

机器翻译 (Machine translation, MT) 是计算语言学的研究领域之一, 其使用计算机软件将文本或语音从一种语言转换为另外一种语言。

从古代的甲骨文, 到现代的语言文字, 正是有了对语言的理解和翻译, 而减少了人类间交流的障碍。

实际上, 翻译的背后是语言, 语言的本质是符号, 因此翻译可以被理解为对符号的“编码”和“解码”的过程。从一种语言翻译到另一种语言, 可以认为对一套符号体系进行编码, 然后再重新解码为另一套符号体系 (如图 8-13 所示, 从甲骨文、现代汉语、现代汉语拼音、英文的编解码翻译方式)。

人	男	女	子	夫	妻	王	口	力	中	又
rén	nán	nǚ	zi	fu	qī	wáng	kǒ	lì	zhōng	yòu
person	man	woman	child	husband	wife	king	mouth	strength	middle	also
目	耳	心	日	月	山	雨	田	好	肉	出
mù	ěr	xīn	rì	yuè	shān	yǔ	tián	hǎo	ròu	chū
eye	ear	heart	sun	moon	mountain	rain	field	good	meat	to go out

图 8-13 甲骨文、现代汉语、现代汉语拼音、英文的翻译。从图中可以看出, 最初的甲骨文为形象字体, 看上去就像一种特殊的符号标记, 现代汉语像是对甲骨文进行解码后得到的, 同样对现代汉语进行编码得到现代汉语拼音, 再解码得到了英文

机器翻译系统中最具代表性的是 Google 翻译 (如图 8-14 所示)。2017 年 3 月 29 日, Google 翻译经过大幅度的优化之后, 宣布在中国正式上线, 重返中国市场。回顾 Google 翻译从 2007 年上线至今, 经历了 10 多年的发展, 已经能够对全球 100 多种语言进行翻译。尽管翻译的质量与专家翻译还有一段距离, 但是可以毫无疑问地说 Google 翻译是全球最好用的翻译软件之一。

早期的机器翻译系统基于单词和语法规则, 那时的神经科学并不如现今的神经科学发达, 人类并不了解自己的大脑是如何处理文字语言的, 但是每一个人都能够

很好地理解和运用语言信息。例如我们每天都在使用中文，对中文言很熟悉，但这不代表我们能够理解中文的语言体系是怎么运作的，这就导致依赖基于单词和语法规则的人工翻译软件漏洞百出，引出很多笑话。



图 8-14 Google 翻译界面

早期机器翻译系统的工作原理是由工程师们根据语言学家提出的复杂规则逐一编程，从而实现不同语言文字之间的翻译和转换。

不足的是，这种由人工制定语言规则的翻译系统只能适用于一些短语，或者新闻标题等结构简单的文字翻译，对于真实世界的文字来说并不可靠。实际上，人类语言并不总是遵循固定的规则，而是充满了各种特殊情况，区域之间表达也有差异，甚至有些人的思维和说话方式都是不遵循套路的。例如如今的英语语法，是受数百年来不断地侵略、殖民影响，从而融合了各不同地区、人民的风俗习惯而形成的，不是由某个人或者某个机构经过严格的规章制度制定下来的。

直到 20 世纪 90 年代初，IBM 发明的统计机器翻译（Statistical Machine Translation, SMT）软件 CANDIDE 结束了翻译软件漏洞百出的尴尬局面。它基于统计机器翻译，通过对大量的语料信息进行统计分析，找出最为常见的词语组合规则，通过统计的方式尽可能避免出现一些奇怪的和不常使用到的短语组合。SMT 翻译短语和短句的效果不错，但是翻译长句子会显得有点吃力，直到近年来基于神经网络机器翻译（Neural network Machine Translation, NMT）崛起，使得翻译长难句的效果得到了进一步提升。

统计机器翻译 SMT 聚焦于句子的局部信息（词组），而神经网络机器翻译 NMT 则更擅长利用句子的全局信息。在对整句信息进行解码、编码后，通过基于深度学习的神经网络模型，在大量的语言文本中学习到高维的语序模型特征，可以发现长句语言模式中嵌套的模式，最终使得长句翻译的效果比以往更好。

### 8.4.1 基于统计的机器翻译

神经网络机器翻译 NMT 与统计机器翻译 SMT 的核心思想都是对源语言进行编



码，然后解码得到目标语言。从 SMT 发展到 NMT 都离不开数理统计的思想原理，因此本节我们来简单地回顾统计机器翻译 SMT 的原理。

现在我们把机器翻译 MT 看成是一个信息传输的过程。假设需要被翻译的语言为源语言  $S$  (Source Language)，想要得到的翻译语言为目标语言  $T$  (Target Language)。假设源语言  $S$  是由一段目标语言  $T$  经过特殊的编码得到的，解码过程则是把源语言  $S$  还原成目标语言  $T$ 。

根据 Bayes 规则可以得到统计机器翻译的基本方程式 (Fundamental Equation of Statistical Machine Translation)：

$$\hat{T} = \operatorname{argmax}_T P(T | S) = \operatorname{argmax}_T P(T)P(S | T) \quad (8-14)$$

其中， $\hat{T}$  是最终的翻译结果； $P(T)$  是目标语言  $T$  出现的概率，称为语言模型，使用目标语言的语料库进行训练； $P(S|T)$  是由目标语言文本  $T$  翻译成源语言文本  $S$  的概率，通过平衡语料库训练得到，因此称为翻译模型。式 (8-14) 的表达如图 8-15 所示。

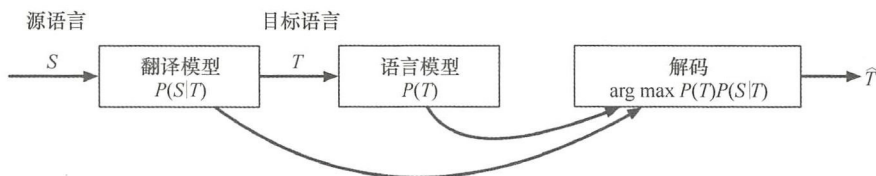


图 8-15 统计机器翻译基本公式组成

值得注意的是，语言模型  $P(T)$  只与目标语言  $T$  相关，与源语言  $S$  无关，反映的是句子在目标语言  $T$  中出现的可能性，实际上表达的是该句子在句法语义等方面的合理程度。而翻译模型  $P(S|T)$  与源语言和目標语言相关，表达的是两个句子互为翻译的可能性。

举个例子，把源中文语言“我有八块腹肌”翻译成目标英文语言“I have eight abdominal muscles”，那么我们需要根据英文语料库计算出  $P(en)$ ，并根据平衡语料库计算出  $P(cn|en)$ ，最后计算基本方程式  $\max P(en)P(cn|en)$ 。了解完基于统计的机器翻译核心思想后，下面我们来了解 SMT 的具体步骤。

### 1. 语料预处理——分词 (Participle)

语料即语言文本，语料预处理就是对语言文本进行分词，加上输入与输出的结果是双语分词后的文件。

### 2. 词对齐 (Alignment)

词对齐的目标是把源语言中的单词或者短语与目标中的单词或者短语对齐。在实际情况中，这是一个非常困难和巨大的工程。例如“北京大学”，具体指的是“北京的大学”，还是一个专业名称。诸如此类的歧义数不胜数，这给词对齐带来了许多

挑战, 如果我们不是语言学家, 又想做好一个词对齐表, 可不是一两个月可以解决的事情。

词对齐最终会转换成为词对齐矩阵的形式, 其中  $i$  代表词对齐矩阵列中元素 (源语言),  $j$  代表词对齐矩阵行中元素 (目标语言), 因此矩阵表达形式分为 4 种:

- $i$  与  $j$  双向对齐;
- $i$  对齐  $j$ ;
- $j$  对齐  $i$ ;
- $i$  与  $j$  没有对齐。

如图 8-16 所示, 图 8-16 (a) 中的 eight 对应中文“八”和“块”, 属于源语言中的一词对应目标语言中的多词, abdominal muscles 对应中文的“腹肌”, 属于源语言的多词对应目标语言的一词。

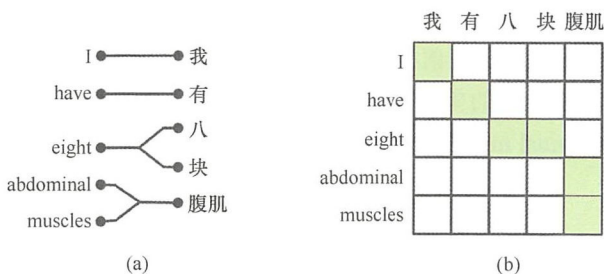


图 8-16 词对齐方式。(a) 为源语言与目标语言的对齐方式, (b) 为词对齐矩阵

### 3. 目标短语抽取 (Phrase Extraction)

短语抽取的目标是抽取出源语言  $S$  与目标语言  $T$  中有可能合并的短语对、词组。由于每一个短语都有很多种翻译的可能性, 因此会产生大量的短语, 最后的翻译会在很大的一个空间结构进行搜索。如图 8-17 所示, 例如中文“我”对应的英文有 (I, I am, I' am), 每一个短语或者单对应目标语言都有多种翻译的可能性。

严格来说, 在第 3 步短语抽取过程中同样也可以算作广义上的词对齐。因为在第 2 步词对齐中, 输出结果是一个词对齐矩阵 (bit 矩阵), 1 代表对齐, 0 代表不对齐。第 3 步则是在第 2 步中生成的 bit 矩阵的基础上进行拓展对齐。

### 4. 计算目标短语概率 (Phrase Probability)

第 4 步是对第 3 步抽取出来满足一致性的目标短语对的出现概率进行估计:

$$p(s|t) = \frac{\text{count}(s,t)}{\text{count}(t)} \quad (8-15)$$

其中,  $\text{count}(t)$  为短语对在目标语言中出现的次数,  $\text{count}(s,t)$  为短语对在源语言和目标语言的匹配次数, 最后式 (8-15) 估计的结果就是翻译模型  $p(s|t)$ 。如图 8-17 所示,

短语抽取完成之后需要计算目标短语概率，英文单词后面的数值为最后的概率得分。

我	有	八	块	腹肌
<u>I</u> (0.3)	<u>has</u> (0.2)	<u>eight</u> (0.4)	<u>pieces</u> (1)	<u>abdominal muscles</u> (0.6)
<u>I</u> am(0.1)	<u>have</u> (0.5)	<u>eighth</u> (0.1)		<u>abdominal</u> (0.2)
<u>I</u> am(0.1)	<u>has</u> (0.2)	<u>august</u> (0.0)		<u>muscles</u> (0.1)
<u>I</u> (0.2)	<u>have</u> (0.3)	<u>eight</u> (0.5)		<u>abdominal muscles</u> (0.1)
<hr/>				
<u>I have</u> (0.2)		<u>eigt pieces</u> (0.3)		
<u>I has</u> (0.3)		<u>eigt</u> (0.8)		
<u>I had</u> (0.1)		<u>eigt piece</u> (0.1)		
...				

图 8-17 基于统计机器翻译的短语概率计算：下划线为对目标语言短语抽取的结果，不仅对单词进行短语抽取，还会进行下一级的短语抽取，短语抽取完之后会计算目标短语概率

### 5. 解码 (Decoder)

解码的目的是在第 4 步产生的目标短语概率基础上，搜索最大的概率作为最优的可能结果，最后通过最大熵计算，对目标短语进行调序。搜索过程如图 8-18 所示，下划线为步骤 4 中计算目标短语概率的过程，图中下半部分为得到的目标短语概率，我们从句子开头搜索最大的概率 I have，然后到 eight 和 abdominal muscles，最终找到 I have eight abdominal muscles 作为搜索结果。

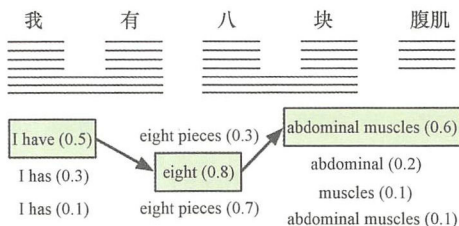


图 8-18 基于统计机器翻译的短语概率计算：下划线为对目标语言短语抽取的结果，不仅对单词进行短语抽取，还会进行下一级的短语抽取，短语抽取完之后会计算目标短语概率

## 8.4.2 基于神经网络的机器翻译

基于神经网络的机器翻译 NMT 思想正如本节与开篇所述，核心思想就是借鉴如图 8-19 所示的编码 - 解码 (Encoder-Decoder) 架构。编码 - 解码与我们思考翻译的过程类似，当大脑听到英文后，想把英文翻译成中文，那么大脑会把英文编码成一堆神经信号  $Z = [z_1, z_2, \dots, z_d]$ 。经过片刻地思考后，就能够把英文句子翻译成中文句子，上述思考过程相当于解码阶段。

在本例中，我们将会使用循环神经网络架构来实现一个编码 - 解码网络模型来完成机器翻译工作。

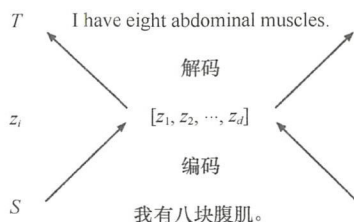


图 8-19 编码 - 解码 (Encoder-Decoder) 架构

### 8.4.3 编码-解码模型

图 8-19 只是编码 - 解码 (Encoder-Decoder) 神经网络模型的简化版，为了深入代码原理，我们需要从具体的输入形式到输出形式对编码 - 解码网络架构进行展开。在进入细节之前，先来看如图 8-20 所示的循环神经网络的编码 - 解码网络架构。该网络架构建立在循环神经网络之上，因为语言文本具有时序性，因此使用循环神经网络更加适合其数据特征表示。整个模型架构由两个循环神经网络连接而成，一个代表编码阶段对源语言进行编码的循环神经网络模型，另外一个代表解码阶段对编码信号进行解码操作的循环神经网络模型。

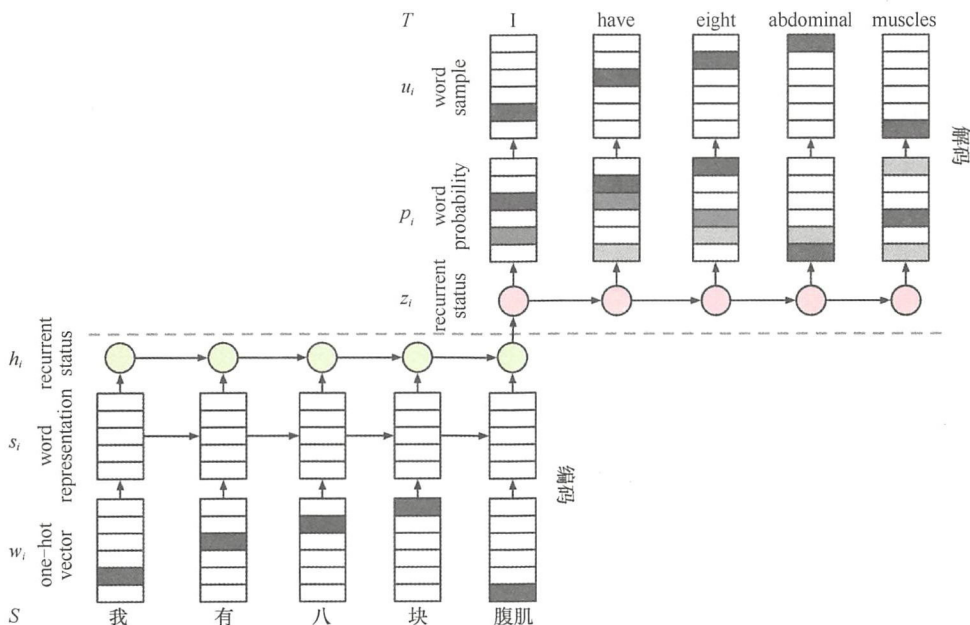


图 8-20 编码 - 解码神经网络详细架构图



## 1. 编码 (Encoder)

图 8-21 所示为编码阶段，圆圈代表 LSTM 的记忆单元，我们可以简单地把它看作一个只有输入、没有输出的 LSTM 循环神经网络，输入带有序列信息的源语言句子  $S$ ，按时间序列通过 LSTM 循环神经网络进行计算，直至到达循环神经网络的最后一个状态  $h_T$  完成对源语言的编码，结束对源语言句子的计算。

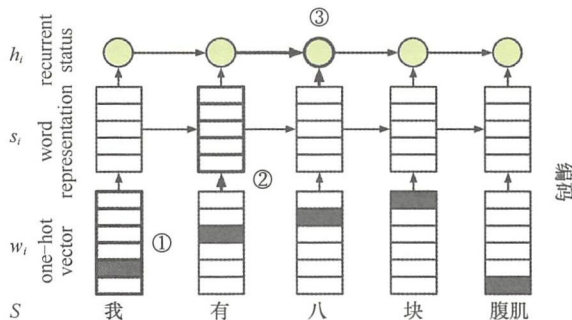


图 8-21 编码 Encoder 阶段循环神经网络架构

神经网络的架构设计很重要，但是千万不要忽略了数据输入的重要性，有时数据预处理阶段可能会占用实际工程中 80% 的时间。接下来我们继续以图 8-21 为例，从数据的输入开始讲解。

① 输入数据采用 one-hot 向量编码方式（该编码具体方式见第 7 章），如果词表有 1000 个单词，那么每一个单词在 one-hot 向量中会有其所属对应的位置。（毫不夸张地说，使用 one-hot 编码可能是处理文本语言中最简单也是效率最低的方式。因为在 one-hot 向量中，每一个单词与其他单词都是等距的，在庞大的单词向量中，仅在单词自身所在的索引位置设置标志位 1，意味着单词跟单词之间不存在任何的关系。）

② 把 one-hot 向量进行词嵌入（Word Embedding）技术操作。Word Embedding 的作用就是通过矩阵  $E$  把 one-hot 向量的稀疏矩阵  $w_i$  转换成为稠密矩阵  $s_i$ ，投影公式为  $s_i = Ew_i$ 。

③ 将 Embedding 后的向量  $s_i$  作为 LSTM 的输入，并根据时间序列进行传播计算。经过 LSTM 网络后，循环网络的最后一个状态  $h_T$  就是将源语言“我有八块腹肌”序列信息进行记忆汇总。

通过上述 3 个步骤，即可把源语言  $S$  成功地编码成循环神经网络的记忆状态。

图 8-21 中的隐层状态  $h_i$  实际上隐藏了很多细节，现在我们先来定义编码阶段中 LSTM 网络所需要的超参数：每一个时间步隐层的数量 input\_depth，每一层隐层的维度 input\_dim，还有为了防止过度拟合的 dropout 概率。具体如【代码清

单 8-3】所示。

### 【代码清单 8-3】编码阶段超参数定义

```
# super-parameter 超参数的定义
dropout = 0.5

input_depth = 1
input_dim = 128
output_depth = 1
output_dim = 128

depth = (input_depth, output_depth)
hidden_dim = (input_dim, output_dim)
```

为了方便，我们在【代码清单 8-4】的运行过程中省略了编码阶段的 Embedding 层，直接使用 one-hot 向量作为输入，因此输入层 `sequence_input` 的输入张量大小为 `[batch_size, sentence_len, source_vocab_size]`。定义完输入层后就到编码层，对 `depth[0]` 进行迭代，对应 LSTM 网络隐层数量的增加。

### 【代码清单 8-4】编码阶段网络模型的定义

```
# 输入层
sequence_input = Input(shape=(sen_len, source_vocab_size), dtype='float32', name='Input')

# Encoder 层
encoder = LSTM(hidden_dim[0], input_shape=(sen_len, source_vocab_size),
               return_sequences=True, activation='relu', name='Encoder')(sequence_input)

# 增加 Encoder 隐层
for _ in range(0, depth[0]):
    encoder = LSTM(hidden_dim[0], return_sequences=False,
                  activation='relu', name='Encoder%d' % _) (encoder)
    encoder = Dropout(dropout, name='EnDropout%d' % _) (encoder)
```

## 2. 解码 (Decoder)

编码和解码阶段的循环神经网络是对称的。如图 8-22 所示，解码阶段中的圆圈同样为 LSTM 的记忆单元，这里的输入是编码阶段所有隐层状态的信息汇总，输出则是对应序列中单词的概率。以图 8-22 为例：

① 将解码阶段最后一次产生的记忆信息  $h_T$  作为解码阶段 LSTM 网络的输入，该记忆信息称为 Thought Vector，这里记作  $c = h_T$ 。然后计算解码阶段 LSTM 的隐层状态  $z_t$ ，其中  $z_t = \varphi(c, z_{t-1})$ ，意味着解码的隐层状态  $z_t$  需要以上一个时间步  $t-1$  和解码阶段的记忆信息 Thought Vector 为依据。

② 根据隐层状态  $z_t$ ，使用 Softmax 函数归一化估算出所属单词的概率  $p_t$ 。

③ 现在得到第  $i$  个时间步的目标单词概率分布，我们可以简单地从这个概率分布中采样得到该目标单词  $u_i$  对应词表的位置，最后查找目标单词词表得到目标语言句子  $T$ 。

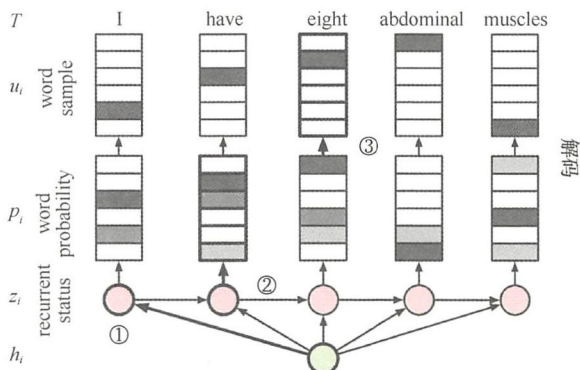


图 8-22 解码阶段循环神经网络架构

在【代码清单 8-5】中，Thought Vector 为编码阶段所有隐层状态的信息汇总，通过使用 RepeatVector 重复目标语言的句子序列作为解码阶段的输入。定义完输入层后就到解码层，对 depth[1] 进行迭代，对应 LSTM 网络隐层数量的增加。

#### 【代码清单 8-5】解码阶段网络模型的定义

```
# thought vector
thought_vector = RepeatVector(target_sen_len, name='C')(encoder)

# Decoder 层
decoder = LSTM(hidden_dim[1], return_sequences=True,
               activation='relu', name='Decoder')(thought_vector)

# 增加 Dncoder 隐层
for _ in range(0, depth[1]):
    decoder = LSTM(hidden_dim[1], return_sequences=True,
                  activation='relu', name='Decoder%d' % _)(decoder)
    decoder = Dropout(dropout, name='DeDropout%d' % _)(decoder)

# 使用 softmax 归一化估算出所属单词的概率 p_i
preds = TimeDistributed(Dense(target_vocab_size, activation='softmax'))(decoder)
```

通过 Python 代码定义了编码 - 解码循环神经网络架构后，现在我们通过【代码清单 8-6】来完成该神经网络架构模型剩余的代码，使用 compile() 函数生成该模型，利用 Adam 优化算法，损失函数则使用交叉熵损失 (categorical\_crossentropy loss)，

并输出模型的具体信息，如图 8-23 所示。

Layer (type)	Output Shape	Param #
Input (InputLayer)	(None, 15, 2000)	0
Encoder1 (LSTM)	(None, 15, 128)	1090048
Encoder0 (LSTM)	(None, 128)	131584
EnDropout0 (Dropout)	(None, 128)	0
C (RepeatVector)	(None, 20, 128)	0
Decoder1 (LSTM)	(None, 20, 128)	131584
Decoder0 (LSTM)	(None, 20, 128)	131584
DeDropout0 (Dropout)	(None, 20, 128)	0
time_distributed_5 (TimeDist	(None, 20, 2000)	258000
Total params: 1,742,800.0		
Trainable params: 1,742,800.0		
Non-trainable params: 0.0		
None		

图 8-23 编码 - 解码神经网络架构输出的信息

#### 【代码清单 8-6】编译机器翻译网络

```
# 模型编译
model = Model(input=sequence_input, output=preds)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# 输出编码 - 解码架构信息
print(model.summary())
```

### 8.4.4 平衡语料数据集

定义完编码 - 解码循环神经网络架构后，现在缺少源语言数据  $S$  和目标语言数据  $T$ 。为了训练上述模型，我们需要一个数据量足够大的平衡语料库，本例中使用 TensorFlow 例子 seq2seq 中使用到的英语 - 法语平衡语料库 WMT'15 Website。

下载 WMT'15 Website 的数据大概需要 2GB 的硬盘空间，解压缩后 fr 法语和 en 英语的二进制文件会占用 8GB 左右的空间。等所有的数据源准备好之后，剩下的就是需要根据目标生成源语言向量  $S$  和目标语言向量  $T$ 。



## 【代码清单 8-7】WMT'15 Website 数据预处理

```
def prepare_vocab(data_dir):
    """ 数据预处理 """
    # 根据定义的词表大小创建词表文件
    source_vocab_path = os.path.join(data_dir, "vocab%d.source" % source_vocab_size)
    target_vocab_path = os.path.join(data_dir, "vocab%d.target" % target_vocab_size)
    create_vocab(source_vocab_path, source_train_path, source_vocab_size)
    create_vocab(target_vocab_path, target_train_path, target_vocab_size)
    print("source_vocab_path:%s finish." % source_vocab_path)
    print("target_vocab_path:%s finish." % target_vocab_path)

    # 创建训练数据对应的句子 token 值
    source_ids_path = os.path.join(data_dir, "token_idx%d.source" % source_vocab_size)
    target_ids_path = os.path.join(data_dir, "token_idx%d.target" % target_vocab_size)
    data2token_ids(source_train_path, source_ids_path, source_vocab_path)
    data2token_ids(target_train_path, target_ids_path, target_vocab_path)
    print("source_ids_path:%s finish." % source_ids_path)
    print("target_ids_path:%s finish." % target_ids_path)

prepare_vocab(basic_path)
```

在【代码清单 8-7】中，函数 `create_vocab` 用来创建词表二进制文件，里面存储着 `vocab_size` 个单词，单词排序按照使用频率，每一行代表一个单词，方便后续对单词进行索引时使用，具体存储方式为 `{b'cat', b'dog', b'bird', ...}`。

而 `data2token_ids` 函数的作用则是根据原平衡语料库中的数据，读取每一行句子，产生由数字组成的 one-hot 编码的句子。如英语句子 “I have eight abdominal muscles”，单词在 `vocab` 二进制文件里的存储序号分别是 “5, 20, 600, 55, 6055, 1000”，经过 `data2token_ids` 函数后，该句子将会变成向量 [5, 20, 600, 55, 6055, 1000] 存储在 `token_idx` 二进制文件中。

如图 8-24 所示，对源数据集进行处理后，得到 `token index` 数据，剩下的就是产生上述网络模型的输入 `encoder_inputs` 和 `decoder_inputs`。其中，源语言数据 `encoder_inputs` 和目标语言数据 `decoder_inputs` 都分别采用了 Padding、Bucketing、Embedding 技术。





Name	^	Date Modified	Size
 token_idx2000.source		May 22, 2017, 15:48	34.6 MB
 token_idx2000.target		May 22, 2017, 15:48	40.7 MB
 vocab2000.source		May 22, 2017, 15:43	16 KB
 vocab2000.target		May 22, 2017, 15:47	17 KB

图 8-24 准备好用于训练的源语言数据和目标语言数据

## 1. padding

在训练之前,我们需要通过填充一些特殊的符号,把训练集中的句子数据转换成固定长度的序列或者可变长度的序列。下面是使用到的特殊符号。

- EOS: End of Sentence, 句子结束标志位。
- PAD: Padding, 补充空格位置标志位。
- GO: Start Decoding, 解码阶段句子开头标志位。
- UNK: Unknown, 不在词典里的单词标志位。

考虑一下源语言  $S$  和目标语言  $T$  这两个句子:

S: 我有八块腹肌

T: I have eight abdominal muscles

如果想要把  $S$ 、 $T$  两个句子转换为固定长度为 10 的序列,最终结果会变成:

S: [ PAD, PAD, PAD, "EOS", "。", "腹肌", "快", "八", "有", "我" ]

T: [ GO, "I", "have", "eight", "eight", "abdominal", "muscles", ".", EOS, PAD ]

在【代码清单 8-8】中,首先定义特殊符号在词表中的序列,方便插入 token 后的句子中:

### 【代码清单 8-8】定义特殊符号

```
# 定义特殊符号
_PAD = b"_PAD"
_GO = b"_GO"
_EOS = b"_EOS"
_UNK = b"_UNK"

# 特殊符号编号
_PAD_ID = 0
_GO_ID = 1
_EOS_ID = 2
_UNK_ID = 3

# 特殊符号索引
_START_VOCAB = [_PAD, _GO, _EOS, _UNK]
_START_VOCAB_ID = [0, 1, 2, 3]
```

对于编码阶段的输入数据 `encoder_inputs`,【代码清单 8-9】中需要在句子后面填充 Padding 的 ID,再反转源语言句子的序号。因为根据 (Sutskever et al.) 论文结果显示,反转输入句子对于神经网络机器翻译能够得到更好的效果。至于解码器的输入,只需要在句子开头加入特殊符号 GO,还有句子末尾补充特殊符号 PAD 即可,不需要句子向量反转。

## 【代码清单 8-9】定义编码阶段和解码阶段的输入

```
# Encoder 的输入 encoder_inputs
# Encoder inputs are padded and then reversed.
encoder_pad = [_PAD_ID] * (encoder_size - len(encoder_input))
encoder_inputs.append(list(reversed(encoder_input + encoder_pad)))

# Decoder 的输入 decoder_inputs
# Decoder inputs get an extra "GO" symbol, and are padded then.
decoder_pad_size = decoder_size - len(decoder_input) - 1
decoder_inputs.append([_GO_ID] + decoder_input + [_PAD_ID] * decoder_pad_size)
```

## 2. Bucketing

在大多数情况下，Padding 填充方法可以改变序列的长度，统一数据的输入。不过随之而来一个更严重的问题，假设数据集里最长的句子有 100 个单词，把所有句子都往后填充 PAD，将句子编码成 100 个序列的长度，以免丢失单词。例如“我有八块腹肌”只有 5 个单词，往后需要填充 95 个 PAD 符号，句子向量中过多无用的信息会掩盖句子中实际的信息。

Bucket 中文翻译为“水桶”，为了避免掩盖句子中的实际信息，同时规整输入的数据，可以考虑把句子放入不同大小的水桶里面。现在有一个 Bucket 的列表 [(5,10), (10,15), (20,25), (40,50)]，如果源句子  $S$  的长度小于 5，那么就把源句子  $S$  和目标句子  $T$  的长度限制为 5 和 10；如果源句子  $S$  的长度大于 5 小于 10，则把  $S$  和  $T$  的长度限制为 10 和 15，依此类推。最后在训练的过程中进行参数共享，即可达到目的。【代码清单 8-10】是 Bucket 为 [(10, 15)] 时：

## 【代码清单 8-10】Bucket 示例

```
# Bucket 为 (10, 15)
S: [ PAD, PAD, PAD, EOS, ".", "腹肌", "块", "八", "有", "我" ]
T: [ GO, "I", "have", "eight", "abdominal", "muscles", ".", EOS, PAD, PAD, PAD,
PAD, PAD, PAD, PAD]
```

## 3. Word Embedding

Word Embedding 层主要作用于编码阶段的输入数据。在没有 Embedding 的情况下，需要把数据转化成为 one-hot 向量。

如【代码清单 8-11】所示，通过 for 遍历 encoder\_inputs 多维向量的每一行（每个 token 句子），然后使用 numpy 的 eye 函数把当前行 tokens 序列数据转换为对应的 one-hot 向量，最后保存在 batch\_encoder\_inputs 向量里。

## 【代码清单 8-11】句子向量 one-hot 编码

```
# batch inputs shape: [total batch size, sentences len, vocabulary size]
```

```

batch_encoder_inputs = np.zeros((train_bucket_sizes, _buckets[0][0], source_
vocab_size), dtype = np.int16)
batch_decoder_inputs = np.zeros((train_bucket_sizes, _buckets[0][1], target_
vocab_size), dtype = np.int16)

# 对 encoder 的输入数据转 encoder_inputs 转换成 one-hot 向量
# Batch encoder inputs are just re-indexed encoder_inputs into numpy format.
print("changing the encoder inputs data into numpy data...")
for _ in range(len(encoder_inputs)):
    batch_encoder_input = np.eye(source_vocab_size)[encoder_inputs[_]]
    batch_encoder_inputs[_] = batch_encoder_input

# 对 dncoder 的输入数据转 dncoder_inputs 转换成 one-hot 向量
# Batch decoder inputs are re-indexed decoder_inputs into numpy format.
print("changing the decoder inputs data into numpy data...")
for _ in range(len(decoder_inputs)):
    batch_decoder_input = np.eye(target_vocab_size)[decoder_inputs[_]]
    batch_decoder_inputs[_] = batch_decoder_input

```

one-hot 向量的优点是简单快捷，但是缺点也是非常明显的，如下。

- 单词与单词之间不存在联系，不能刻画单词与单词之间的关系，无法有效表达语义。
- 严重的数据稀疏性。假如一个句子有 20 个单词，词表有 20000 个单词，那么这个句子的 one-hot 向量是 [20,2000]，有 10 万个等待训练的句子，维数灾难就来了：对带有 1 亿个参数的稀疏矩阵进行训练。为了解决 one-hot 向量的缺点，研究者们发明了 Word Embedding 技术。

Word Embedding 的目标就是要寻找一个新的向量，使得意思相近的单词间距离更短，并且该向量尽可能稠密。例如：句子向量为 [[4],[20]]，经过 Embedding 层后向量变成 [[0.25,0.1],[0.6,-0.2]]。实际上 Embedding 是通过特定的神经网络训练出来的。

图 8-25 所示为对单词句子进行 Word Embedding 操作后降维，抽取部分单词的空间关系进行对比。从图中可以看出，相同或者相近意思的单词会聚束在一起，意思相反的单词反而在空间上距离越远，这就是 Embedding 层的工作目的。如果要深入 Embedding 层可能会耗费不少时间，有兴趣的读者可以进一步阅读（Sebastian Ruder et al., 2016）中关于 Word Embedding 的介绍。

为了增加 Embedding 层，避免直接使用 one-hot 向量，需要对【代码清单 8-5】中的编码 - 解码架构进行少许修改。在【代码清单 8-12】中的输入层不能够直接使用三维向量 [total batch size, sentences len, vocabulary size]，而是使用没有经过 one-hot





的 tokens 向量 [batch size, sentences len]，最后经过 Embedding 层后，数据产生的向量将会变回三维向量 [batch size, sentences len, EMBEDDING\_DIM]。

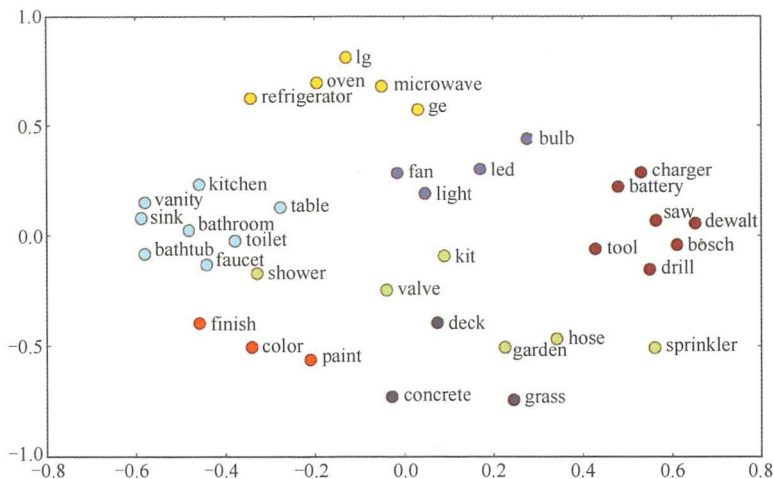


图 8-25 使用 Word Embedding 技术后对单词进行降维后得到单词间的空间关系

其中，one-hot 向量的第三维度为词表的大小，embedding 向量的第三维度 EMBEDDING\_DIM 一般设置为 100 ~ 500，在数据维度上 embedding 占了绝对的优势，同时让语义之间的联系更加紧密。

#### 【代码清单 8-12】修改输入数据为 Word Embedding

```
# 输入层
sequence_input = Input(shape = (input_bucket_size,), dtype='int32')

# Embedding 层
embedded_sequences = Embedding(source_vocab_size, EMBEDDING_DIM,
                               input_length = input_bucket_size, trainable = False)(sequence_input)
```

## 8.4.5 机器翻译的未来

神经网络机器翻译 NMT 并不是能解决各种语言翻译问题的“万能钥匙”。我们也并不是在任何情况下都选择神经网络机器翻译，而摒弃传统的机器翻译方式。现今，越来越多的机器翻译系统，包括世界领先的 Google 翻译系统采用结合统计机器翻译（SMT）与神经网络机器翻译（NMT）一起工作，为用户提供更加精准的翻译结果。

神经网络机器翻译从技术角度看是一个显著的进步，但从语言角度来看却并非



如此。从基于规则的机器翻译和基于统计的机器翻译应用过程中来看,某些语言根本不适合使用机器来翻译。例如,一些已经失传的古代语言“古埃及语言”,这些语言根本不适合采用机器翻译。语言学家需要去推测、根据历史事件去猜想某个符号背后所代表的深刻含义,在遥远的古代,那是集合了众多智者的力量,提炼出来的最具有代表性的符号。又例如,在缺乏大量语料数据的前提下,谈神经网络或者概率都言之过早。相信未来会有更多的开源平衡语料库喷涌而生,大数据驱动着人工智能的发展,人工智能反过来也驱动着的大数据的发展。

虽然神经网络及其翻译存在着与生俱来的缺点,但是伴随着深度学习的到来,机器翻译的准确率节节攀升,这就仿佛预示着未来翻译行业的改变和新一轮的洗牌,每一位翻译从业者都会被深度学习带来的价值所惊讶。可以肯定地说,机器翻译在可以预见的未来能够取代大量的人工翻译。现在的人工翻译市场跟机器翻译的市场有各自的针对应用场景和应用群体:人工翻译对准高端市场,要求精准的翻译需求;而机器翻译针对的翻译场景的要求没有人工翻译高,对翻译的精度没有太苛刻的要求,可以用在旅游、网页浏览、信息监控等领域。未来,以人工智能为代表的机器翻译将会进一步提高其翻译准确率,完成更多如今看上去不可能实现的任务。

## 8.5 示例2: 智能对话机器人

还记得2013年初人人网上那只“贱贱的”、能够自动回复的小黄鸡吗?有没有留意在淘宝中向店小二打个招呼“有人吗?”,瞬间就能得到一个回复“亲,在的”?当我们不想用手机直接操作时,只需要语音呼叫iPhone上的Siri,她就会出来为我们提供语音互动(如图8-26所示)。众多的例子都是因为自动问答机器人的存在。

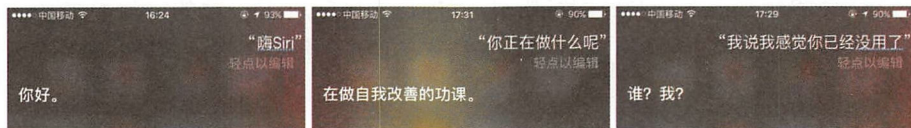


图 8-26 iPhone 中的 Siri 是一个自动问答机器人,能够辅助用户进行功能操作,帮助用户搜索资讯、普通聊天等

大多数科技的诞生,其核心价值都是服务于商业目的,自动问答机器人也不例外。现今,随着深度学习的发展,带动着商业模式向前探索,自动问答机器人已经不仅仅用于在线聊天了。商业的自动问答机器人可以部署在许多应用媒介上,如移动设备、社交媒体、消息应用、语音响应和自动短信回复。自动问答机器人可以预



测客户的意图，收集客户的信息，深入地了解用户需求，减少企业客服数量，增加企业的对外服务吞吐量。

## 8.5.1 Seq2Seq模型

在没有使用深度学习之前，实现一个自动问答机器人需要大量定向的语料库，然后经过切词、反向索引、建立索引词表、词典加载，对用户输入的问题句子进行预处理、歧义消解，最后通过最大匹配算法等一系列自然语言处理相关的算法操作，获得最终的答案。然而得到的答案还可能会因为在处理数据过程中某一步操作不当，导致与最终的结果差别很大，但难以跟踪是哪一个环节算法引起的问题。

(Oriol Vinyals et al., 2014) 提出了类似于机器翻译的循环神经网络模型 Seq2Seq，利用该网络我们可以实现一个自动问答系统。如图 8-27 所示，Thought Vector 的左边是编码阶段，对序列数据“Am I handsome boy?”送入循环神经网络，输入的序列数据作为问题；Thought Vector 的右边是解码阶段，对编码得到的 Thought Vector 进行解码，得到序列数据“you make me really sick”作为问题的回答内容。

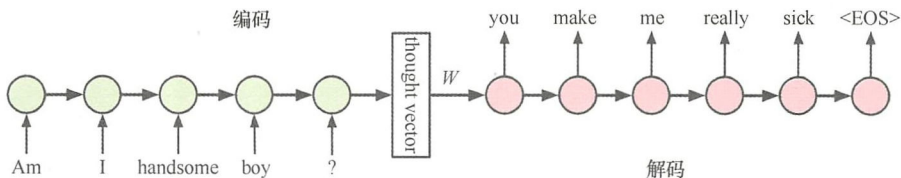


图 8-27 编码 - 解码模型框架

对于上一节中神经网络机器翻译的编码 - 解码架构，有的人也称之为序列到序列 (Sequence to Sequence) 的神经网络架构，实际上它们两个是一样的，为了方便起见，在本节中我们会使用简称 Seq2Seq 模型来统称类似于的编码 - 解码一类的循环神经网络架构。

在本例中，我们使用 Cornell Movie--Dialogs Corpus 中提供的数据作为问答机器人的数据源。该语料库从电影中提取大量的对话数据，共提取 10292 个电影人物的 220579 个对话，该电影对话数据集足够产生一个效果不错的问答机器人。

在 Cornell Movie--Dialogs Corpus 数据集中，原始数据是 movie\_lines.txt 和 movie\_conversations.txt 两个文件，其具体的组织形式请参考 README.txt。这里的代码与机器翻译的代码几乎一模一样，区别在于对问答数据的提取。

【代码清单 8-13】是作者训练的结果，左边是问题 Q，右边是答案 A（实际的训练





效果并不是每一句回复都非常理想，下面是截取其中较好的一部分结果作为展示)：

【代码清单 8-13】问答机器人训练后的结果

```
Q: hi, jack.
A: Yes.

Q: What are you up to?
A: What do you want to know.

Q: Nothing, I am so good.
A: Me too.

Q: Are you busy?
A: Can I help?
```

## 8.5.2 Seq2Seq模型的缺点

基本 Seq2Seq 网络模型的应用如上所述已经很广泛，但不代表其效果是最好的。实际上，在介绍基本 Seq2Seq 网络模型时，为了方便理解，对该模型做了一些规约简化，下面来探讨一下基本 Seq2Seq 网络模型的缺点。

### (1) 缺少输入与输出数据的联动等细节的处理

在传统自然语言任务的处理中，需要对序列信息进行切分、建立词表等操作。上述操作过程中需要处理很多序列信息上的细节，并有针对性地对序列数据进行统计和存储，如果有特殊需求，还需要人为地去筛选、修改部分数据。而 Seq2Seq 模型则缺少很多细节的处理，循环神经网络可以根据损失函数的定义学习网络中的参数，但当前时间节点  $t$  的损失函数只涉及之前时序的记忆信息，计算出记忆向量 Thought Vector 后，后面的解码阶段就没有再与输入的序列信息进行沟通。

### (2) 需要更大的网络模型结构才能存储和表达更长的序列数据

对于基本的 Seq2Seq 网络模型，编码阶段把输入的序列数据压缩为固定大小的向量，解码阶段则把该固定大小的向量转换为目标序列数据。其中，该固定大小的向量 (Thought Vector) 必须要包含源输入序列数据信息的大部分细节，最后需要让编码贴近真正的函数—非线性且足够复杂，能够表达源序列信息。最后在实际的工程处理中，Thought Vector 的维度需要足够大，来存储更长时间序列数据的高维特征。

### (3) 受制于 GPU 的存储空间

更大的网络模型和更长的序列信息意味着 GPU 需要更多的计算量和内存空间。假设我们使用 NVIDIA 最新的 GPU，确实能够提升模型的计算量，但是并不代表着拥有更多的显存空间（至少现在还没能做到 GPU 的内存空间同计算量一样不断地增加，一般板载的 GPU 显存为 8GB）。因此，我们不能一直向 GPU 要更多的资源，而





应该在有限的资源下完成更多的事情。

(4) 输入信息的时间序列越长，预测效果的准确率越低

如图 8-28 所示，随着句子长度的增加，Seq2Seq 模型对于目标序列的预测效果呈现下降的趋势。

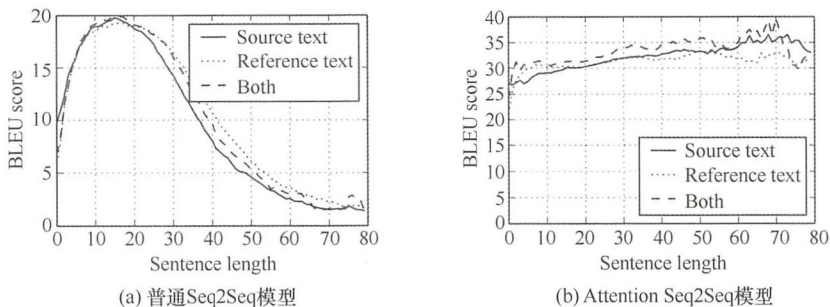


图 8-28 基本 Seq2Seq 模型与加入 Attention 机制模型对于处理机器翻译不同长度序列数据的表现

### 8.5.3 超越Seq2Seq框架

在学习了神经网络机器翻译的编码 - 解码网络模型后，我们已经深入了解其内部工作方式。基于神经网络机器翻译 NMT 任务，输入序列数据  $S$  可以是中文，输出序列数据  $T$  可以是英文；对于文本摘要任务，输入序列数据  $S$  可以是一篇文章，输出序列数据  $T$  可以是该文章对应的摘要；对于自动问答系统任务，输入序列数据  $S$  可以是问题，输出序列数据  $T$  可以是答案。对于使用 Seq2Seq 网络模型的应用场景还有很多，涉及两个序列模型需要转换、提取时，都可以思考能否使用 Seq2Seq 网络模型进行处理。

Seq2Seq 网络模型的一般提升方法是：在编码和解码阶段的循环神经网络，可以使用最基本的循环神经网络模型、LSTM、GRU、双向循环神经网络 (Bi-directional RNN)、Deep LSTM 等升级架构，使用不同的循环神经网络处理不同的序列数据，可能会产生我们意想不到的结果。另外，我们还可以在编码阶段使用卷积神经网络，解码阶段使用循环神经网络，提取图像中的序列信息作为图像生成模型。

接下来，我们在 Seq2Seq 模型的基础上加入一些机制 (Feedback 机制、Peek 机制、Attention 机制)，其目的是尽量减少因为 Seq2seq 模型的缺点引起的精度下降，并且希望在有限的内存上提高网络模型的信息熵，让其存储更大量的信息，最终使得网络模型能够处理更加复杂的任务。



## 1. FeedBack 机制

在基本 Seq2Seq 网络模型中, 解码阶段循环神经网络在时间步  $t$  只把  $t-1$  的状态信息作为输入, 这样的方式能够非常好地学习到目标序列信号的序列信息。如图 8-29(a) 所示, 在解码阶段当前时间序列  $t$  的状态计算公式为:

$$z_i = \sigma(z_{i-1}) \quad (8-16)$$

为了让 LSTM 网络模型学习到输出的信息, 把在解码阶段的时间步  $t-1$  输出的序列信息  $t_i$  也作为当前时间序列  $t$  状态的输入。如图 8-29(b) 所示, 当前的时间序列  $t$  的状态计算公式为:

$$z_i = \sigma(z_{i-1}, t_i) \quad (8-17)$$

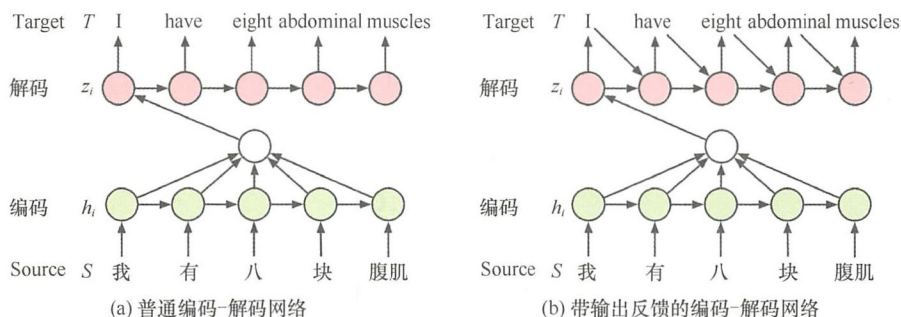


图 8-29 基本 Seq2Seq 网络模型与带反馈 Feedback 网络模型

## 2. Peek 机制

Feedback 机制作用是把上一个时间步  $t-1$  的输出反馈到下一个时间步  $t$  的隐层状态进行计算, 其计算公式为  $z_i = \sigma(z_{i-1}, t_i)$ 。对应图 8-29(b), 现在来分析带 Feedback 的解码阶段循环隐层状态的公式:

$$\begin{aligned} z_1 &= \sigma(C, t_0) \\ z_2 &= \sigma(z_1, t_0) \\ z_3 &= \sigma(z_2, t_1) \\ &\dots \end{aligned} \quad (8-18)$$

其中,  $C$  为编码阶段最后产生的记忆向量, 仅仅作作为解码阶段的首次隐层状态输入, 这样的操作在解码阶段的隐层状态  $z_i$  计算中, 忽略了编码提取的信息。为了避免该问题, 加入 Peek 机制后, 编码阶段提取的向量  $C$  也作为解码阶段的隐层状态  $z_i$  的输入, 如图 8-30 所示, 计算公式为:

$$z_i = \sigma(z_{i-1}, t_{i-1}, C) \quad (8-19)$$



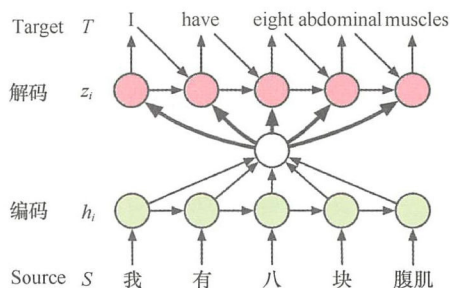


图 8-30 带 Peek 机制的 Seq2Seq 网络模型

### 3. Attention 机制

带 Peek 机制的 Seq2Seq 网络架构模型并没有加入注意力 (Attention) 机制, 其解码阶段隐层状态计算方式为:

$$\begin{aligned} z_1 &= \sigma(z_0, t_0, \underline{C}) \\ z_2 &= \sigma(z_1, t_1, \underline{C}) \\ &\vdots \\ z_i &= \sigma(z_{i-1}, t_{i-1}, \underline{C}) \end{aligned} \quad (8-20)$$

编码阶段最后的隐层状态信息存储在向量  $\underline{C}$  中。根据式 (8-19),  $\underline{C}$  作用于解码阶段的全部隐层状态。因此可以把 Seq2Seq 网络模型架构视为注意力不集中的模型 (或者没有注意力的模型)。

举个简单的例子, 句子“我有八块腹肌”, 实际上我们最关心的不是八块还是六块腹肌, 只要不是一块, 都可以证明你真的拥有肌肉。在基本 Seq2Seq 网络模型中,  $\underline{C}$  作编码阶段的编码信息传递给每一个解码器的隐层状态, 这就会导致有用的和没用的信息都会往后传递。对于例句“我有八块腹肌”, “我”和“腹肌”的权重应该最大, 这样才能体现出源句子中当前单词对于目标单词的影响程度不同, 类似给出下面一个概率分布值:

$$(\text{我}, 0.4), (\text{有}, 0.05), (\text{八块}, 0.05), (\text{腹肌}, 0.5)$$

编码阶段每一个时间步  $t$  实际上是有各自的权重值, 基本 Seq2Seq 网络模型无法体现该情况。即该 Seq2Seq 网络模型注意力不集中, 对网络模型的不同输入缺乏注意力机制, 统一认为网络的输入都具有相同的权重值。实际上, 对于短序列来说, 没有引入 Attention 时效果不是很明显。但是如果输入的序列句子较长, 此时所有语义仅通过一个中间隐层向量  $\underline{C}$  来表示, 单词自身的信息本来已经被编码压缩过, 如果没有加入 Attention 信息, 最终会丢失更多的细节, 或给网络引入更多的噪音。图 8-31 所示为加入了 Attention 后, Seq2Seq 网络对“我”“有”“腹肌”集中注意, 传给后续解码的隐层。



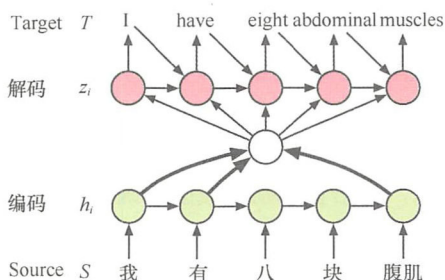


图 8-31 带 Attention 的 Seq2Seq 网络模型

加入 Attention 的好处在于，网络中有了 Attention 后，不再需要将完整的源序列编码为固定长度的向量  $C$ 。相反，我们允许编码解码器在每一个时间步  $t$  的输出都选择性地参与到编码向量的不同部分。很重要的一点是，Seq2Seq 模型会自动地根据输入的序列信息与记忆单元中的门控机制决定什么时候参与网络的计算，什么时候不参与网络的计算。Seq2Seq 网络解码阶段得到的编码向量返回到输入序列中，而不是把所有的输入序列都编码成固定长度的向量，使得相同大小的网络模型文件中可以存储更多信息，进而提高网络模型的信息熵，使 Seq2Seq 网络模型能够处理更加复杂的任务。

## 8.6 示例3: 智能语音识别音箱

自动语音识别 (Automatic Speech Recognition, ASR) 可谓是人工智能的重要入口。从京东和科大讯飞合作推出的“叮咚”、Amazon 的明星产品“Echo”、Google 的“Master”和百度“小度”音箱的相继推出，到阿里巴巴人工智能实验室“天猫精灵”入局，互联网巨头掀起的新人机交互大战，赚足了各大媒体和用户的眼球（如图 8-32 所示）。



图 8-32 智能音箱。从左到右分别是 Amazon 的 Echo、Google 的 Master、Apple 的 HomePod、阿里巴巴的天猫精灵

自动语音识别技术正在慢慢地渗入我们的生活，新一代的游戏主机、智能手表、智能手机等智能终端设备已经开始内置语音识别程序。在 2017 年的“双十一”天猫





购物节上，只需要 99 元就可以买到一个让我们在家里用语音购物、查询天气、选择音乐或者有声读物的智能音箱。我们甚至不用翻开手机查询快递信息，只要大声地说出需求，智能音箱就能告诉我们相关的快递资讯。

语音识别技术已经出现了 20 多年，为何近年来才成为人工智能的主流技术呢？这要得益于深度学习技术，将语音识别领域的准确率提高到足以应用于实际环境中。接下来让我们一起去了解如何使用深度学习进行语音识别吧！

## 8.6.1 语音识别框架

自动语音识别（ASR）技术是一种让机器通过识别和理解，把人类的语音信号转变为相应文本的技术过程。

早在 20 世纪 90 年代初期，就已经出现众多语音识别领域的研究人员试图利用人工神经网络 ANN 进行自动语音识别方面的研究，可是大部分效果并不理想，原因如下：

- 语音数据有限；
- 神经网络容易过拟合；
- 计算资源有限等。

而与此同时，基于概率论的技术在语音识别领域得到蓬勃的发展，例如高斯混合模型（Gaussian Mixture Model, GMM）、隐马尔科夫模型（Hidden Markov Model, HMM）等。

经过 20 多年的发展，自动语音识别技术已经发展成一个固定的框架结构，该模型主要分为编码（Encoder）和解码（Decoder）阶段，具体如图 8-33 所示。

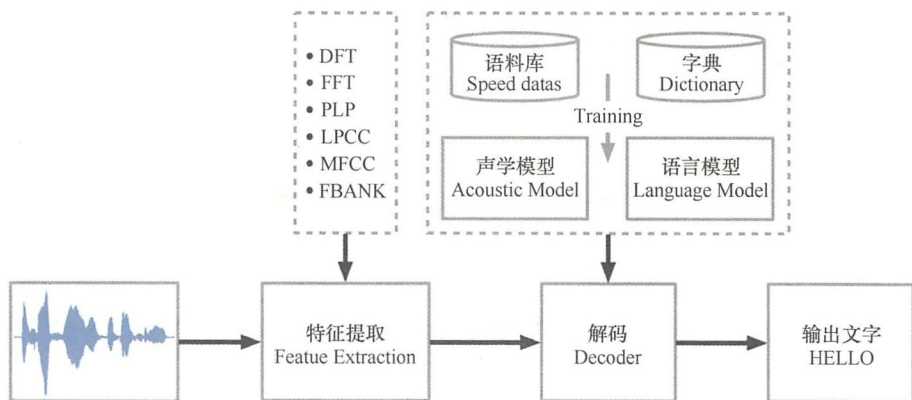


图 8-33 自动语音识别（ASR）系统的一般架构。从最基本的语音信号数据作为 ASR 系统的输入，对语音进行特征提取得到声纹特征，到通过声学模型（Acoustic Model）和语言模型（Language Model）对声纹特征进行解码，最终输出该语音信号数据对应的文本文字



- **编码**: 将音频数据作为输入, 转换成音频向量数据。
- **特征提取**: 通过算法或者音频特征算法提取音频向量, 提取后的特征称为“声纹”。例如使用快速傅里叶变换 (Fast Fourier Transform, FFT) 对音频数据进行时域和频域间的转换。
- **训练**: 训练阶段是从声纹数据和字典数据中学习固定特征, 用于生成声学模型 (Acoustic Model) 和语言模型 (Language Model)。其中, 声学模型用于识别语音向量, 一般可以使用 GMM 或者循环神经网络等方法来识别向量, 用 HMM 或者 CTC 来对齐输出的结果; 语言模型是根据语法、语义规则对声学模型调整输出的结果, 例如修改与调整不符合逻辑规则的词语。
- **字典**: 在语音识别领域中大部分模型并不是以单词作为基本单位, 而是以音素作为基本的语音识别单位。
- **解码**: 将训练好的声学模型和语言模型进行组合, 输入新的声纹特征, 最终输出其对应的文本文字。

### 音素 (Phone)

音素是语音中最小的单位, 依据音节里的发音动作来分析, 一个动作构成一个音素。音素分为元音和辅音两大类, 英语辅音和元音在语言中的作用就相当于汉语中的声母和韵母。例如英文单词 HELLO ['hʌ'ləu] 中有 5 个音素。

图 8-33 所示的自动语音识别 (ASR) 系统的一般架构仍然有点复杂, 至少以本书目前介绍的知识暂时无法完全掌握自动语音识别所使用到的技术。本节关心的是使用深度学习技术进行语音识别, 因此我们会实现一个简单的声学模型 (如图 8-34 所示)。从简单的音频数据开始, 对其特征提取得到“声纹”, 通过循环神经网络模型实现一个声学模型, 最后解码输出该音频数据所对应的文本文字。

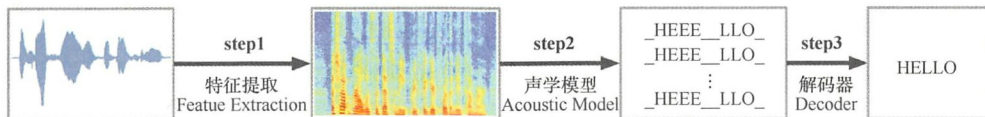


图 8-34 本例的语音识别流程。原始音频数据作为输入, step1 对音频数据进行特征提取, step2 使用声学模型对声纹特征进行处理, step3 对得到的数据进行解码, 最终输出该音频数据对应的文本文字

不同的人会有不同的语速, 说话方式和行为也会不一样。例如, 一个人可能会带有疑问地说出 [HEEEEEELLO?], 而另外一个人则可能很开心地说出 [HELLOOOOOOOOOOO!], 这样对应同一个单词会产生不同长度的声音文件。而语音识别的任务就是把上面两个声音文件都正确识别为 [HELLO]。实践证明, 把各种不同长度的音频文件自动对齐到一个固定长度的文本是一件很困难的事情。

为了解决上述问题，我们会使用一些特殊的技巧，例如在特征提取阶段进行对齐操作或者在输出阶段对音素进行对齐操作等。接下来我们先会学习与深度学习无关，但是对于语音识别至关重要的技术。

## 8.6.2 准备语音数据

语音识别的第一步是将声波输入计算机中。声音以波的形式进行传播，我们如何将声波转换为数字形式呢？

图 8-35 所示为作者说出的 [HELLO] 的声音片段，声波在每一个时间都有对应的波幅。为了将声波转换为数字，我们以等距的方式将声波隔开，记录下声波在等距点的高度，称为比特率（数字声音由模拟格式转化成数字格式的采样率）。

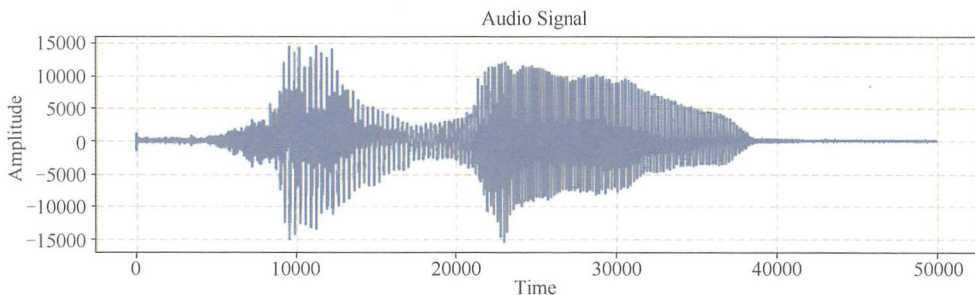


图 8-35 [HELLO] 的声音片段

有了声波的比特率之后，在规定时间间隔内对声波进行采样（Sampling），假设每秒读取数万次（以达到硬件物理限制），并把声波在这个间隔时间内的波幅值记录下来，得到一个未压缩无损的音频文件。例如标准 CD 格式就是 44.1kHz 的采样频率（每秒采样 44100 次）。

由于人的发声频率在 100Hz（男低音）～10000Hz（女高音）范围内，正常人能够听见 20 Hz～20000 Hz 的声音，而老年人的高频声音减少到 10000Hz（或可以低到 6000Hz）左右。因此，对于语音识别系统，我们只需要 1.6kHz 的采样频率（每秒采样 16000 次）就足以覆盖人类语音的频率范围。

下面以 openslr 中的音频数据集为例，下载 dev-clean.tar.gz 和 test-clean.tar.gz 作为训练数据集和验证数据集。如【代码清单 8-14】所示，将下载的数据集中无损压缩的 flac 文件转换为 1.6kHz 的采样频率、64k 比特率的 wav 文件。

### 【代码清单 8-14】音频文件格式转换

```
for flacfile in `find . -iname "*.flac"`  
do
```

```
avconv -y -f flac -i $flacfile -ab 64k -ac 1 -ar 16000 -f wav "${flacfile%.*}.wav"
done
```

将无损压缩的音频文件转换为符合一定采样率要求的音频文件后，在【代码清单 8-15】中读取系统中数据集的文件数据路径，并分别把训练集和验证集的音频数据路径 key、对应的音频时间 duration 和音频对应的英文文本 test 记录下来，方便后续训练时对音频进行检索。

#### 【代码清单 8-15】生成训练集和验证集对应文件

```
def handler(data_dir, output_file):
    """ 读取音频文件 """
    sentences = [] # 用于记录音频对应的英文文本
    durations = [] # 用于记录音频时间
    keys = [] # 用于记录音频路径

    # 读取给定路径下所有子目录
    for group in os.listdir(data_dir):
        group_path = os.path.join(data_dir, group)

        # 读取子目录下所有说话者的数据
        for speaker in os.listdir(group_path):
            labels_file = os.path.join(group_path, speaker,
                                       '{}-{}.trans.txt'.format(group, speaker))

            # 检索音频对应的英文文本
            for line in open(labels_file):
                split = line.strip().split()
                file_id = split[0]
                sentence = ' '.join(split[1:]).lower()
                sentence = get_valid_str(sentence)
                audio_file = os.path.join(group_path, speaker, file_id)
                audio = wave.open(audio_file + '.wav') # 打开音频文件
                duration = float(audio.getnframes())/audio.getframerate() # 获取音频时长
                audio.close()

                keys.append(audio_file)
                durations.append(duration)
                sentences.append(sentence)

    # 存储记录信息到制定文件中
    with open(output_file, 'w') as out_file:
        for i in range(len(sentences)):
            line = json.dumps({'key':keys[i], 'duration':durations[i], 'text':sentences[i]})
            out_file.write(line + '\n')
```



经过上面的代码操作后，获得如下所示的文件存储格式。

```
{"key": "/test-clean/1089/134686/1089-134686-0000.wav", "duration": 5.435,
"text": "he hoped there would be stew for mutton flour fattened sauce"}
```

```
{"key": "/test-clean/1089/134686/1089-134686-0001.wav", "duration": 3.275,
"text": "stuff it into you his belly counselled him"}
```

在【代码清单 8-16】的 AudioHandler 类主要实现对验证集数据和验证集数据文件的读取，并把相关的数据存储在内存中。

#### 【代码清单 8-16】读取训练集文件和验证集文件

```
class AudioHandler():
    def __init__(self, batch_size=20, feature_type='spectrogram'):
        """ 定义相关的参数 """
        self.step = 10
        self.window = 20
        self.max_freq = 8000
        self.eps = 1e-14
        self.train_index = 0
        self.valid_index = 0
        self.train_len = 0
        self.valid_len = 0
        self.spec_dim = int(0.001 * self.window * self.max_freq) + 1
        self.batch_size = batch_size
        self.feature_type = feature_type
        self.feature_mean = np.zeros((self.spec_dim,)) # 训练数据特征的平均值
        self.feature_std = np.zeros((self.spec_dim,)) # 训练数据特征的方差值

    def load_train_data(self, file=None):
        """ 读取训练集 """
        paths, times, texts = self._load_data_from_file(file)
        self.train_len = len(paths)
        self.train_paths = paths
        self.train_times = times
        self.train_texts = texts

    def load_test_data(self, file=None):
        """ 读取测试集 """
        paths, times, texts = self._load_data_from_file(file, split=200)
        self.valid_len = len(paths)
        self.valid_paths = paths
        self.valid_times = times
        self.valid_texts = texts
```

```
def _load_data_from_file(self, file):
    paths = []
    times = []
    texts = []
    with open(file) as json_file:
        for i, data in enumerate(json_file):
            spec = json.loads(data)
            if float(spec['time']) > 10.0: continue # 跳过音频时间过长数据
            paths.append(spec['path'])
            times.append(spec['time'])
            texts.append(spec['text'])

    return paths, times, texts
```

准备好训练集数据和测试集数据之后, 调用 `AudioHandler` 类, 给出指定的音频数据索引, 见【代码清单 8-17】, 就可以得到对应的音频数据 (如图 8-36 所示)。

#### 【代码清单 8-17】调用 `AudioHandler` 类并显示声波图

```
>>> loading_path = "zomi/data/LibriSpeech/"
>>> train_path = os.path.join(loading_path, "train_corpus.json")

>>> audio_gen = AudioHandler()
>>> audio_gen.load_train_data(train_path)
>>> raw_audio = audio_gen.train_paths[666]
>>> plot_raw_audio(wave.read(raw_audio))
```

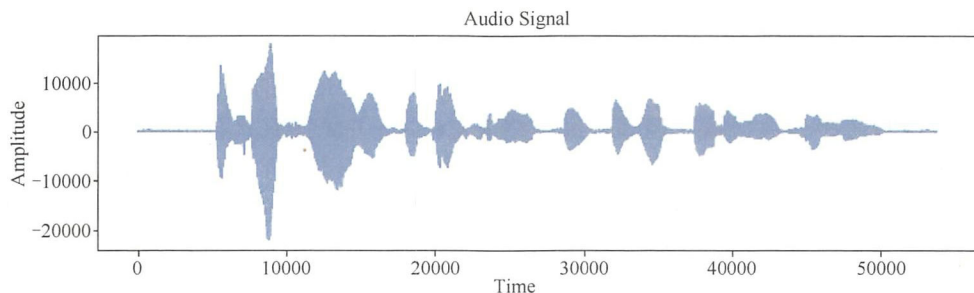


图 8-36 索引对应测试集的音频数据, 英文文本为 [a shop girl was the destiny prefigured for the newcomer]

回到声波的知识里, 把图 8-15 中 [HELLO] 的声波以每秒 16000 次的采样率、比特率设置为 64k 来显示前 100 个采样数据。其中, 每个数据表示声波在 1/16000 处的振幅, 也就是声波对应比特率的值。

```
array([ 2,  2,  6, -29, -80, -102, -86, -56, -29, -16, -1,
       -11, -12, -24, -26, -35, -28, -12,  8,  17, -10, -27,
```

```

-31, -11, -27, -29, -24, -12, -30, -50, -81, -72, -20,
.....
-28, -10, -36, -25, 23, 4, -66, -61, 4, 26, -24,
-67, -30, 25, 21, -16, -61, -78, -79, -48, -24, -83,
-66, -17, -15, -13, -31, -67, -91, -96, -93, -81, -59])

```

由于声波的采样频率是对原始声波间歇性地读取。如图 8-37 所示，左图为原始声波，右图对声波进行间歇性的读取值，某种程度上可以认为是对原始声波进行粗略地近似估计。如此的采样方式与真实的音频数据之间必然就会造成部分数据丢失。

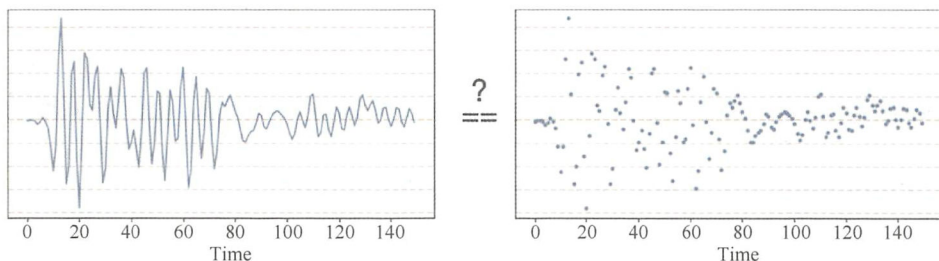


图 8-37 音频采样方式

既然数字模拟采样的方式不能够完美重现原始声波，那应该怎么处理这些时间上的间距呢？一个很直观的方法是利用采样定理（Nyquist Theorem），使用更高的采样率，获得更好的音频质量，因此通过提高比特率和采样频率来近似还原原始声波。

然而在实际处理过程中，使用更高的采样率并不一定能够获得更好的效果。假设现在有一个时间序列，序列中每一个值代表 1.6kHz 中的声波振幅值，我们可以把该时间序列数据输入循环神经网络 RNN 中，并试图直接分析时间序列的采样数据来进行语音识别。可是经过实际工程测试，这样的方式很难完成有效的语音识别。相反，我们可以通过对音频数据进行一些预处理来使问题变得更加简单。

### 8.6.3 语音特征提取

首先把音频分成每份 20ms 长的音频块（window=20），即对应音频上的一帧数据。假设以每秒 16000 次的采样频率，那么一个 20ms 的音频对应如下所示 320 个采样数据。对该 320 个采样数据进行图形化显示，得到在 20ms 内的声波的大致形状如图 8-38 所示。

```
array([ 109, 106, 95, 75, 75, 50, -1, -33, -68,
       -113, -144, -174, -199, -198, -190, -194, -212, -249,
       ...,
       172, 217, 208, 140, 131, 195, 218, 197, 196,
       205, 205, 223, 250, 243, 222, 226, 209, 164], dtype=int16)
```

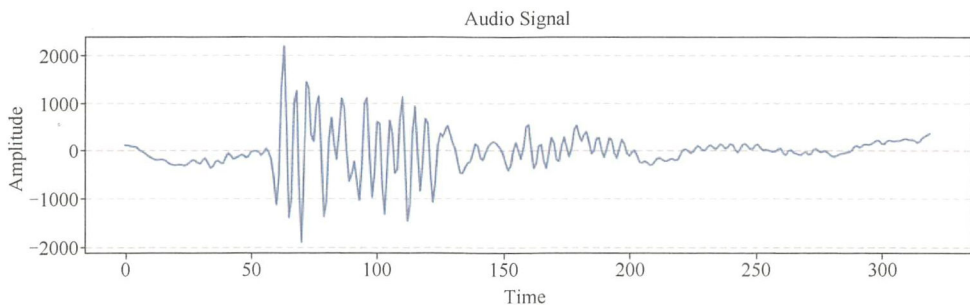


图 8-38 音频采样片段

图 8-38 所示的音频帧虽然只有短短的 20ms，但即使较短的音频片段也是由不同频率的声音交织而成，其中包括低音、中音和高音。正是这样不同频率的声音组合，才有了我们听到的音频数据文件。

接下来，为了使得音频数据更加容易地被循环神经网络处理，我们把一段连续的音频声波分解成很多段短暂的音频采样片段，例如上述例子所示的以 20ms 为间隔对音频进行切片采样。

这里引入一个问题，语音识别的一个重要特点是待识别的语音内容是不定长时序的。也就是在进行语音识别之前，我们不知道当前一个音素有多长，对声学模型输入语音特征时，我们很难判断输入的一个时间片段应该截取为 20ms 还是 50ms，才能更好地包含一个音素，从而给声学模型进行识别；另外一个原因是大部分声学模型都不方便处理维度不确定的输入特征。

针对上述问题，一个简单的解决方法是依然按照固定间隔对音频进行切片采样。假设切片的时间间隔内足够长，以包含大部分音素（即声学模型足以判断该时间片段内是属于哪个音素），然后间隔时间内有一定程度上的重叠。在实际工程中，以 step 来表示每次移动的时间序列步长，window 来表示每次截取的时间片段长度。另外，还可以引入相邻时间片之间的差分作为额外的特征等。

按照上述原理，将声波分解成由多个时间段组成的音频采样片段，然后将该音频采样片段的低音部分分离出来，接着在该音频片段下继续分离最低音部分，依次类推。完成以上操作后，我们从音频采样片段中得到不同的频段（Frequency Band），



把相同频段中的能量值相加，得到该音频采样片段的声纹理（Voiceprint）。

这一操作步骤可以由傅里叶变化（Fourier Transform）来实现，傅里叶变化能够将复杂的声波分解为简单的声波。一旦有了这些单独的声波，将每一份频段所包含的能量值相加，就能形成新的音频片段特征。

以快速傅里叶变化（FFT）为例，最终得到的音频特征表示的是从低音到高音每个频段范围的重要程度。继续以 20ms 的音频所包含的能量值从低频到高频进行数值输出，如下所示。对采样数据进行图形化显示，得到如图 8-39 所示的声纹特征。

```
array([-24.78862351, -27.02052672, -24.81015279, -25.01879259,
       -25.13204948, -24.78210174, -24.08304256, -25.83264403,
       -27.50418836, -24.31194649, -24.79711438, -23.22395698,
       -23.70121185, -23.16847955, -25.06526124, -24.67108899,
       .....
       -25.8126652 , -27.24670389, -25.85342664, -25.03371072,
       -24.01247867, -23.34999352, -24.71004333, -24.28349919,
       -23.553137 , -22.40590023, -22.59734261, -22.29045457,
       -22.68754961, -22.81121927, -23.3259266 , -22.22534451])
```

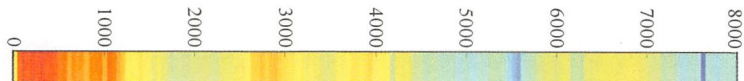


图 8-39 经过快速傅里叶变换（FFT）得到的片段声纹

在一段音频数据中，每次移动 10 个时间序列、截取 20 个实践片段（step=10，window=20）进行快速傅里叶变化，最终可以获得如图 8-40 所示的频谱图。在【代码清单 8-18】中实现归一化处理 `normalize()` 函数，目的是对声纹特征数据进行归一化处理，进而更好地输入给神经网络进行处理。另外有关 MFCC 特征转换代码和 FFT 特征转换代码可以参考本书的 GitHub 代码库。

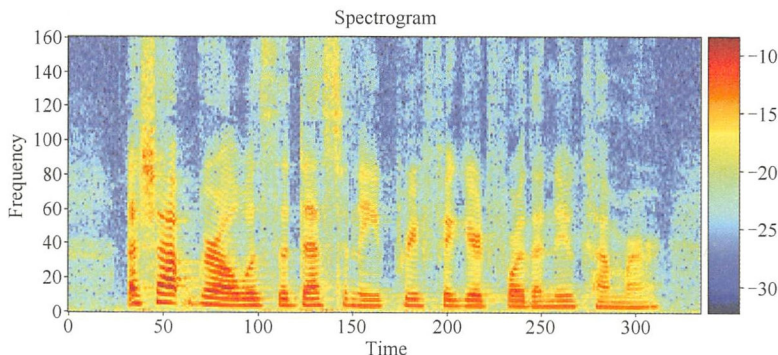


图 8-40 快速傅里叶变化（FFT）频谱图

【代码清单 8-18】声纹特征转换部分代码

```

def normalize(self, feature):
    """
    对特征进行归一化处理
    """
    return (feature - self.feature_mean) / (self.feature_std + self.eps)

def featurize_mfcc(self, wave_path):
    """
    MFCC 特征转换 .
    """
    return FeatureMFCC(wave_path)

def featurize_normal(self, wave_path):
    """
    FFT 特征转换
    """
    return FeatureSPEC(wave_path, max_freq=self.max_freq, step=self.step,
window=self.window)

AudioHandler.normalize = normalize
AudioHandler.featurize_mfcc = featurize_mfcc
AudioHandler.featurize_normal = featurize_normal

```

有了基本特征的转换代码后，下面来分别显示经过归一化和没有经过归一化的数据频谱图，见【代码清单 8-19】。从图 8-40 的右侧栏中可以看出，没有经过归一化的数据分布在  $[-10, 30]$  之间，在图 8-41 的右侧栏中，经过归一化后的数据分布在  $[-2, 3]$  之间，并以 0 为对称轴分布，因此可以证明经过归一化的数据更符合普遍的规律，便于神经网络的处理（与归一化相关的更多知识可以参考第 3 章）。

【代码清单 8-19】显示 FFT 频谱图

```

def plot_voice_feature(mfcc_feature, name):
    """ 显示频谱图 """
    fig = plt.figure(figsize=(12, 5))
    ax = fig.add_subplot(111)
    im = ax.imshow(mfcc_feature, cmap=plt.cm.jet, aspect='auto')
    plt.title(name)
    plt.ylabel('Frequency')
    plt.xlabel('Time')
    divider = make_axes_locatable(ax)
    cax = divider.append("right", size= "5%", pad = 0.1)
    plt.colorbar(im, cax = cax)

```

```

ax.set_xticks(np.arange(0, 13, 2), minor = False);
plt.show()

>>> spect_feature = audio_gen.featurize_normal(raw_audio)
>>> norspec_feature = audio_gen.normalize(spect_feature)
>>> plot_spect_feature(tr_spec_feature, "Spectrogram") # 显示经过 FFT 转换得到的频谱图
>>> plot_spect_feature(tr_norspec_feature, "Normalized Spectrogram") # 显示对归一化后的频谱图

```

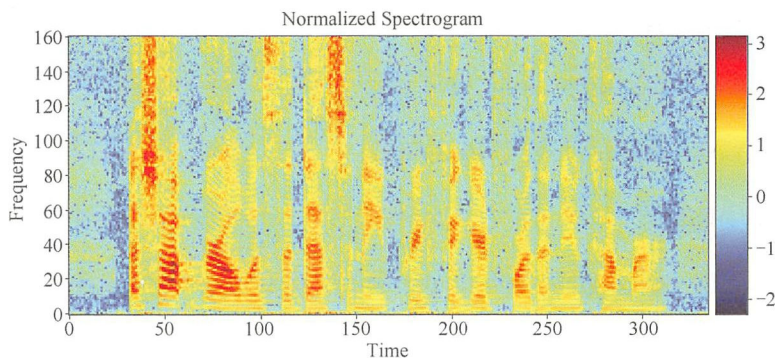


图 8-41 归一化后的快速傅里叶变化 (FFT) 频谱图

梅尔频率倒谱系数 (Mel-Frequency Cepstral Coefficients, MFCC) 中的梅尔频率是基于人耳听觉特性提取出来的, 利用梅尔频率与音频频率成非线性对应关系, 计算得到频谱特征。其主要用于语音数据特征提取, 降低声学模型的运算维度, 见【代码清单 8-20】。例如: 对于一帧有 320 采样点的数据, 经过 MFCC 后可以提取出最重要的 13 维数据, 提取音频数据特征的同时也达到了降维度的目标。

#### 【代码清单 8-20】显示 MFCC 频谱图

```

>>> mfcc_feature = audio_gen.featurize_mfcc(raw_audio)
>>> tr_normfcc_feature = audio_gen.normalize(mfcc_feature)
>>> plot_spect_feature(mfcc_feature, "MFCC") # 显示经过 MFCC 转换得到的频谱图
>>> plot_spect_feature(tr_normfcc_feature, "Normalized MFCC") # 显示对归一化后的频谱图

```

如图 8-42 和图 8-43 所示, 从 FFT 频谱图和 MFCC 频谱图中可以看出音频数据中的高音和低音模式, 更加直观地了解音频数据的音频能量分布情况。对于神经网络而言, 与原始声波相比, 不仅时间序列被缩放到一个固定的序列长度, 另外从频谱数据中能够更加容易地找到规律。因此, 对于声学模型的输入, 我们使用的是经过特征提取后的频谱数据。

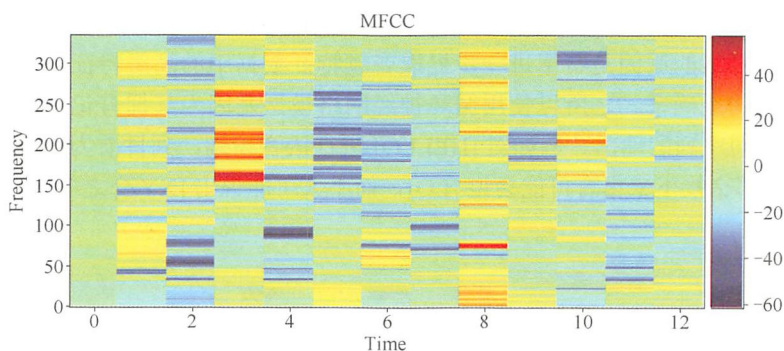


图 8-42 梅尔频率倒谱系数 (MFCC) 频谱图

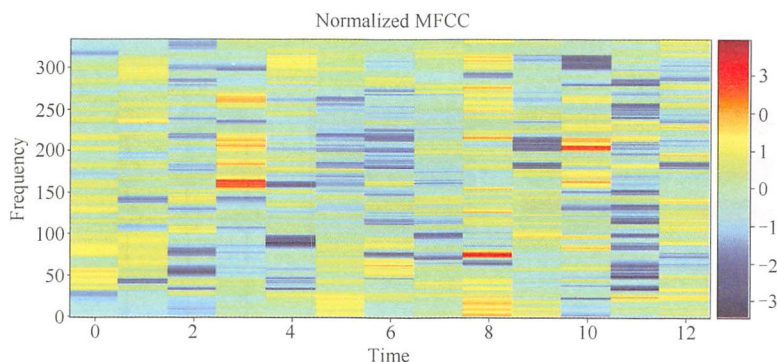


图 8-43 归一化梅尔频率倒谱系数 (MFCC) 频谱图

## 8.6.4 声学模型

声学模型 (Acoustic Model) 用于识别语音向量, 这里采用深度神经网络作为声学模型。现在有了在格式上是易于神经网络模型处理的音频特征数据, 可以将其作为深度神经网络的输入。深度神经网络的输入是以 20ms 为单位的一帧, 每一帧作为一个时间序列, 深度网络尝试找到当前音素对应的字母。

使用 [HELLO] 的音频文件经过声学模型的前馈计算, 可以得到每一帧音频对应的英文字母, 如图 8-44 所示。(为了方便实现, 本例中声学模型的输出使用了英文字母 a ~ z 作为音素的代替。)

### 1. 模型定义

根据 (Amodei et al., 2016), 我们使用卷积神经网络模型对每一帧音频特征进行高维特征提取, 然后把提取到的高维数据作为循环神经网络模型的输入, 循环神



神经网络模型使用 3 层的 GRU 网络模型如图 8-45 所示，代码见【代码清单 8-21】。使用卷积神经网络的原因是希望进一步提取音频特征数据中的高维特征，提高循环神经网络模型的准确率。之所以使用循环神经网络模型是因为其拥有记忆功能，用于影响未来时间序列的输出。例如对 [HELLO] 进行语音识别，到目前为止我们已经说了 [HEL]，那么接下来很有可能会说 [LO]，而不太可能说出 [XYZ] 之类的音素。

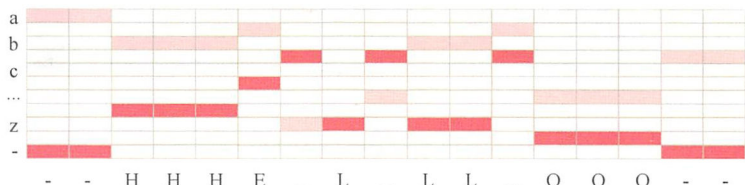


图 8-44 声学模型的输出

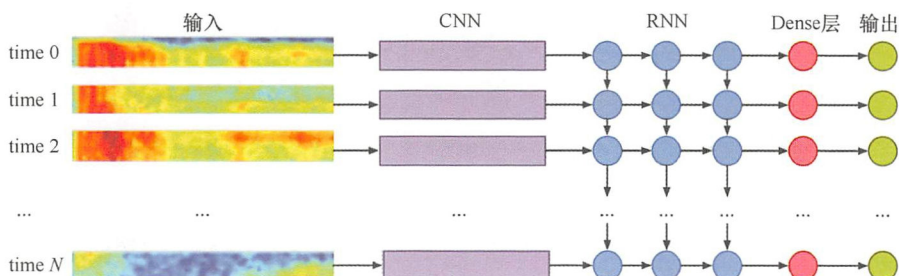


图 8-45 卷积神经网络与循环神经网络结合的声学模型。该声学模型使用 FFT 作为音频提取特征，每一个时间维度的频谱数据作为声学模型的时间序列输入；经过一维的卷积神经网络后，接一个 3 层的循环卷积神经网络，经过 Time Dense 层后输出

### 【代码清单 8-21】声学模型

```
def gru_model(input_dim=161, output_dim=29, recur_layers=3, nodes=1024):

    # 定义参数
    kernel_size=11                                # 卷积核大小
    initialization='glorot_uniform' # 初始化方式

    # 输入层
    input_data = Input(shape=(None, input_dim), name='the_input')

    # 卷积层
    conv_1d = Conv1D(nodes, kernel_size, padding='valid', strides=2,
                      kernel_initializer=initialization, activation='relu', name='conv1d')(input_data)
```

```

# BN 层
output = BatchNormalization(name='bn_conv1d')(conv_1d)

# RNN 层
for i in range(recur_layers):
    # GRU 层
    output = GRU(nodes, activation='relu', kernel_initializer=initialization,
                  return_sequences=True, name='rnn_{}'.format(i+1))(output)
    # BN 层
    otuput = BatchNormalization(name='bn_rnn_{}'.format(i+1))(output)

# 输出层 (Softmax)
time_dense = TimeDistributed(Dense(output_dim))(otuput)
y_pred = Activation('softmax', name='softmax')(time_dense)

model = Model(inputs=input_data, outputs=y_pred)
model.output_length = lambda x: x
return model

>>> gru_model = gru_model()
>>> print(gru_model.summary())

```

Layer (type)	Output Shape	Param #
=====		
the_input (InputLayer)	(None, None, 161)	0
conv1d (Conv1D)	(None, None, 1024)	1814528
bn_conv1d (BatchNormalizatio	(None, None, 1024)	4096
rnn_1 (GRU)	(None, None, 1024)	6294528
rnn_2 (GRU)	(None, None, 1024)	6294528
rnn_3 (GRU)	(None, None, 1024)	6294528
bn_rnn_3 (BatchNormalization	(None, None, 1024)	4096
time_distributed_2 (TimeDist	(None, None, 29)	29725
softmax (Activation)	(None, None, 29)	0
=====		

```
Total params: 20,736,029
Trainable params: 20,731,933
Non-trainable params: 4,096
```

## 2. 加载数据

在机器翻译系统和自动问答系统的例子中，我们并不需要手动地指定训练集或者验证集中每一个 batch 的内容，因为其输入和输出文本数据经过转换后得到的向量（如 one-hot 向量）能够直接存储在有限的内存中，提供给神经网络调用训练。而本章中声学模型例子使用大量的语音数据（动辄需要几十甚至是几百 GB 的硬盘空间），并且需要对语音数据进行特殊的特征提取才能作为网络的输入。因此对于声学模型输入数据，我们要进行分批加载和分批特征转换。

【代码清单 8-22】中的 `get_batches()` 函数将指定 batch 中的数据内容分批加载到内存中，训练时调用 `next_train()` 函数和 `next_valid()` 函数实现加载任务。其中，`index` 记录当前加载的索引号，便于下一次从 `index` 到 `index + batch_size` 的索引位置加载数据。

其中值得注意的是输入对齐问题。由于每一个音频的特征中能量值数量是不固定长度的（假设使用快速傅里叶变换 FFT 进行转换，窗口和步长大小相同下得到相同的时间序列），另外每一个音频的文本长度都是不固定的，因此使用 `max_len` 和 `max_testlen` 来记录最长的数值，把较短的数据特征放入固定大小特征向量中，对于没有数值的向量值使用默认值来填充补齐。

例如在一次 batch 数据中，最短文本为 [hello]，最长文本为 [hello my chicken wings]。以英文字母作为输出的 label，`max_testlen` 为最长文本数 22，把其余位置以“28”进行填充补齐（28 索引为空“-”）。

```
[ h e l l o - - - - - - - - - - - - - - - ]
[ 9 6 13 13 16 28 28 28 28 28 28 28 28 28 28 28 28 28 28 28 ]

[ h e l l o _ m y _ c h i c k e n _ w i n g s ]
[ 9 6 13 13 16 2 14 26 2 4 9 10 14 12 6 15 2 24 10 15 8 20 ]
```

最后输入声学模型的 `data` 和 `label` 分别为音频特征向量和上述的单词向量。`get_batches()` 函数返回 `{input, output}` 字典。`__shuffle_data('train')` 函数则是在一次 epoch 迭代结束之后，把训练集或者验证集的数据打乱并重新排序，使得下一次训练的数据与上一次训练的数据不同，增加了深度神经网络的鲁棒性。

### 【代码清单 8-22】获取训练集数据和验证集数据

```
def get_batches(self, batch_name):
    """
```

对于训练和验证的时候，获得每一个 batch 对应的数据

```
"""
if batch_name == 'train':
    index = self.train_index # index 作为当前 batch 数据的索引
    paths = self.train_paths
    texts = self.train_texts
elif batch_name == 'valid':
    index = self.valid_index # index 作为当前 batch 数据的索引
    paths = self.valid_paths
    texts = self.valid_texts
else:
    raise Exception("Invalid batch name.")

# dim 为数据维度，feats 为获取 batchs 对应的数据，norfeats 为归一化后数据
dim = self.spec_dim
feats = [self.featurize_normal(p) for p in paths[index:index + self.batch_size]]
norfeats = [self.normalize(f) for f in feats]

# 计算输入和输出中数据大小
# max_len 是在 batch 中的特征长度，如：(2939,161)
# max_textlen 是在 batch 中的文字长度，如：(65)
max_len = max([norfeats[i].shape[0] for i in range(0, self.batch_size)])
max_textlen = max([len(texts[index + i]) for i in range(0, self.batch_size)])

# 初始化必要的矩阵
data = np.zeros([self.batch_size, max_len, dim]) # (20, 2939, 13)
labels = np.ones([self.batch_size, max_textlen]) * 28 # (20, 65)
input_length = np.zeros([self.batch_size, 1]) # (20, 1)
label_length = np.zeros([self.batch_size, 1]) # (20, 1)

迭代 batch 中的特征
for i, feat in enumerate(norfeats):
    input_length[i] = feat.shape[0] # 输入网络中的数据长度
    data[i, :feat.shape[0], :] = feat # 记录输入网络中的数据 data

    # 计算输出数据和输出数据的长度
    label = np.array(text2seq(texts[index + i]))
    labels[i, :len(label)] = label
    label_length[i] = len(label)

# keras 神经网络的输入数据
inputs = {
    'the_input': data,
```



```

        'the_labels': labels,
        'input_length': input_length,
        'label_length': label_length,
    }
    # keras 神经网络的输出数据
    outputs = {
        'ctc': np.zeros([self.batch_size])
    }
    # 返回输入输出数据
    return (inputs, outputs)

def next_train(self):
    """
    获得训练集中的 batch 数据
    """
    while 1:
        batchs_result = self.get_batchs('train')

        # 更新训练集中的索引作为下一次 batch 使用
        self.train_index += self.batch_size

        # 边界检查
        if self.train_index >= self.train_len:
            self.train_index = 0
            self.__shuffle_data('train')

        yield batchs_result

def next_val(self):
    """
    获得验证集中的 batch 数据
    """
    while 1:
        batchs_result = self.get_batchs('valid')

        # 更新验证集中的索引作为下一次 batch 使用
        self.valid_index += self.batch_size

        # 边界检查
        if self.valid_index >= self.valid_len:
            self.valid_index = 0
            self.__shuffle_data('valid')

        yield batchs_result

```

### 3. 输出对齐

#### (1) CTC 概述

语音识别声学模型的训练属于监督学习，需要知道每一帧对应的 label 标签才能进行有效的训练。在传统的语音识别声学模型中，在对语音模型进行训练之前，往往要求语音与文本进行严格的对齐操作。例如语音音频 [hello my chicken wings] 采用时间段 5000 ~ 10000 对应文本 hello，时间段 12000 ~ 30000 对应文本 my。依此类推，文本与语音音频需要把采样时间段逐一对齐。

然而在上面例子中的输入对齐代码中，并不是一种严格的对齐方式，而是一种较为宽松的对齐方式。虽然现在已经有了很多较为成熟的开源语音对齐工具可供使用，但是随着深度学习的发展，我们希望让深度神经网络自己去学习对齐的方式，因此连接时序分类（Connectionist Temporal Classification, CTC）应用而生。

CTC 可以理解为基于神经网络的时序分类，目的是在神经网络中将语音音频和文字对应起来，解决时序上输入输出不对称的问题。其带来的好处有两点：

- 不需要将语音音频数据与文本数据进行一一标注对齐；
- CTC 可以直接输出时间序列预测的概率，不需要额外的处理。

具体如图 8-46 所示，输入时序数据经过循环神经网络的处理后，得到 Softmax 的输出，输出的时间序列特征数据并不一定与最终的文本序列一一对称。例如输入的音频特征有 130 个时间窗口，输出对应的文本数据为 [hello my chicken wings] 只有 22 个字母（包括中间空格）。CTC 层通过计算，使得输入与输出对应起来，减少了大量的标注时间，并使得声学模型能够做到端到端的训练。

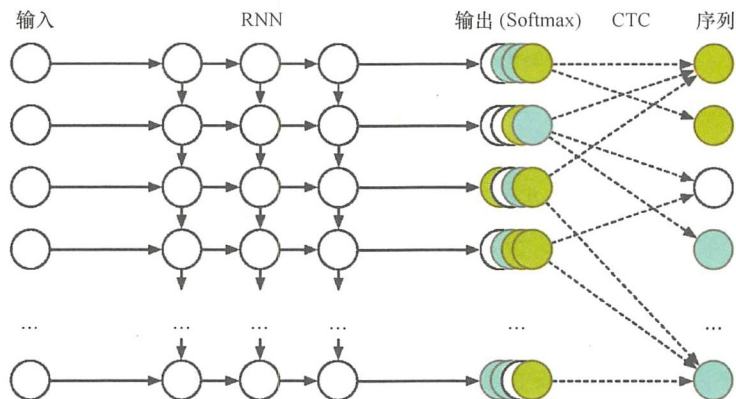


图 8-46 CTC 作用于声学模型的输出层

## (2) 路径和 B 变换

在实际的训练情况中，我们并不知道每一帧对应的音素。下面来考虑一种简单的情况，假设已经知道音频 [hello] 中每一帧音素的标签，其中训练样本为  $x$ ，输出标签为  $z$ 。由于每一帧都会有一个对应标签，因此有：

$$z = \left[ \underbrace{h, h, \dots, h}_{t_1}, \underbrace{e, e, \dots, e}_{t_2}, \underbrace{l, l, \dots, l}_{t_3}, \underbrace{l, l, \dots, l}_{t_4}, \underbrace{o, o, \dots, o}_{t_5} \right] \quad (8-21)$$

在音频 [hellp] 中， $z$  包含音频中每一帧对应的文本标签，因此有：

$$p(z|x) = p(z|y = N_w(x)) = y_{z_1}^1 y_{z_2}^2 \dots y_{z_T}^T \quad (8-22)$$

其中， $N_w$  为循环神经网络模型， $y$  为循环神经网络模型的输出预测值， $y_{z_i}^i$  为在第  $i$  时间帧循环神经网络模型输出为  $z_i$  的概率。我们希望式 (8-22) 的值越大越好，因此上式写成：

$$\min_w \left[ -\log(y_{z_1}^1 y_{z_2}^2 \dots y_{z_T}^T) \right] \quad (8-23)$$

如果训练数据中的每一帧都标记了正确的音素，如式 (8-21) 所示，那么训练过程就会变得简单很多。可是实际上强对齐的数据很少，更多的是没有进行强对齐的数据，而 CTC 可以做到使用没经过逐帧标记的数据进行深度神经网络的训练。

下面我们定义一些英文字符代替音素作为标记，表示所有音素的集合，其中 \_ 为 blank，\_ 为 None：

$$L = \{-, a, b, c, \dots, x, y, z, _\} \quad (8-24)$$

下面用  $\pi$  表示由  $L$  中元素组成长度为  $T$  的路径：

$$\begin{aligned} \pi^1 &= (h, h, h, _, a, a, a, a, _, l, l, l, _, u, u, u, u) \\ \pi^2 &= (h, h, h, _, a, a, a, a, _, l, l, l, l, _, o, o, o, o, o, o) \\ \pi^3 &= (h, h, _, e, e, e, e, e, e, _, l, l, l, l, _, l, l, _, o, o, o) \\ \pi^4 &= (h, h, h, h, h, _, a, a, _, l, _, l, _, u, u, u, u, u, u) \\ \pi^5 &= (h, h, h, _, e, e, e, e, _, l, l, l, l, l, l, _, l, l, l, l, l, _, o, o, o, o, o) \end{aligned} \quad (8-25)$$

式 (8-25) 表示 5 条不同的路径，其中路径  $\pi^3$  和  $\pi^5$  都可以认为是音频 [hello] 所对应的文本文字。

通常来说，标签路径  $\pi$  的长度小于音频数据帧的长度。例如标签序列为 [hello]，训练时输出的序列可能是 [hh\_eee\_ll\_ll\_o]。那怎么建立输出序列与标签序列之间的联系，从而训练整个声学模型呢？



CTC 采用了一个自定义的映射函数  $B$ , CTC 的不同之处在于输出状态引入了一个 blank, 输出和 label 经过  $B$  变换可以将多个输出序列映射到同一个输出:

$$\begin{aligned} B(aaabbbccc) &= abc \\ B(a\_b\_c) &= abc \end{aligned} \quad (8-26)$$

因此如果有一条路径使得  $B(\pi^*) = (h, e, l, l, o)$ , 则认为该路径  $\pi^*$  为最终输出文本。

### (3) CTC 损失

假设每一时刻输出的标签概率互不影响, 那么整体输出的概率就等于各时刻输出概率的乘积, 即路径  $\pi^* = (\pi_1, \pi_2, \dots, \pi_T)$  的概率为其网络预测值  $y$  相乘。以  $x$  作为输入序列, 一种路径的概率为:

$$p(\pi | x) = \prod_{t=1}^T y_{\pi_t}^t \quad (8-27)$$

实际情况中, 多个路径会对应一个正确的序列, 并且这个序列长度往往小于路径长度, 目标函数为  $\{\pi | B(\pi) = z\}$  中所有元素的概率和, 序列最终的概率可以用路径的概率之和来表示:

$$p(l | x) = \sum_{B(\pi)=z} p(\pi | x) \quad (8-28)$$

其中,  $l$  表示对应的标注文本, 我们的目标是通过输入序列  $x$  得到输出序列  $y$ , 如果可以获得输出序列的分布  $p(l|x)$ , 选择其中概率最大的那一个作为输出序列即可 (如图 8-47 所示), 因此有:

$$L(x) = \max_{l \in L} p(l | x) \quad (8-29)$$

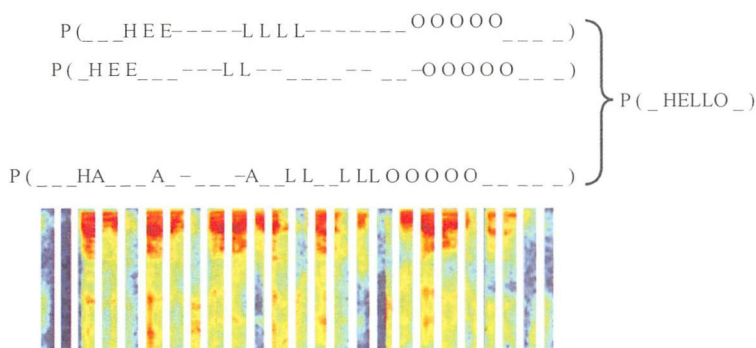


图 8-47 CTC 层对路径概率求最大和得到最终路径结果





路径长度越长，音频对应可能出现的路径越大，路径的总数为  $L^T$ 。因此，最终 CTC 借用了 HMM 中的向前向后算法（Forward-Backward Algorithm）来计算可能路径，见【代码清单 8-23】。向前因子  $\alpha$  和向后因子  $\beta$  的定义为：

$$\alpha(t, u) = \sum_{\pi \in V(t, u)} \prod_{i=1}^t y_{\pi_i}^i$$

$$\beta(t, u) = \sum_{\pi \in W(t, u)} \prod_{i=1}^{T-t} y_{\pi_i}^{t+i} \quad (8-30)$$

向前向后算法通过动态规划的思想来解决，核心思路是针对一个当前标签  $u$  的全部路径累加，被分解为以  $u$  为前缀的全部路径的迭代累加，该迭代通过递归计算向前向后因子求得。有兴趣的读者可以进一步参考相关文献（Rabiner, 1989）。

#### 【代码清单 8-23】调用 TensorFlow 中的 CTC 层

```
def CTC_lambda(args):
    """
    ctc_batch_cost(y_true, y_pred, input_length, label_length)
    """
    y_pred, y_true, input_length, label_length = args
    return K.ctc_batch_cost(y_true, y_pred, input_length, label_length)

def CTC_loss(rnn_model):
    labels = Input(shape=(None,), dtype='float32', name='the_labels')
    input_length = Input(shape=(1,), dtype='int64', name='input_length')
    label_length = Input(shape=(1,), dtype='int64', name='label_length')

    loss_out = Lambda(CTC_lambda, output_shape=(1,), name='ctc')(
        [rnn_model.output, labels, input_length, label_length])

    model = Model(inputs=[rnn_model.input, labels, input_length, label_length],
        outputs=loss_out)
    return model
```

## 4. 网络训练

我们经历了了解语音识别框架，准备语音数据，提取语音数据的特征，建立声学模型，并解决了输入对齐和输出对齐的问题后，就可以开始训练自己的声学模型了。在【代码清单 8-24】中，训练阶段代码与前面的例子中训练代码没有太多差别，网络的优化算法使用第 2 章介绍的随机梯度下降 SGD 算法。值得注意的是，在 `fit_generator()` 代码中，`generator=audio_gen.next_train()` 每个 batch 读取一批数据进入内容提供给网络进行训练，迭代次数 `epochs` 选择 100。



**【代码清单 8-24】声学模型网络训练**

```

loading_path = "zomi/LibriSpeech/"
train_path = os.path.join(loading_path, "train_corpus.json")
test_path = os.path.join(loading_path, "test_corpus.json")

# 加载把训练集和验证集（测试集）
audio_gen = AudioHandler(feature_type='mfcc', batch_size=20)
audio_gen.load_train_data(train_path)
audio_gen.load_test_data(test_path)

# 计算测试集 epoch 内的样本数
train_steps = audio_gen.train_len // batch_size
valid_steps = audio_gen.valid_len // batch_size

# 添加 CTC 损失函数
acoustic_model = CTC_loss(rnn_model)

# 使用随机梯度下降 SGD 算法进行网络优化
sgd = SGD(lr=0.02, decay=1e-6, momentum=0.9, nesterov=True, clipnorm=5)
acoustic_model.compile(loss={'ctc':lambda y_ture, y_pred:rnn_model.output},
optimizer = sgd)
checkpoint = ModelCheckpoint(monitor='val', filepath='result/model_{epoch:02d}.h5')

# 开始网络训练
acoustic_model.fit_generator(generator=audio_gen.next_train(),
                             steps_per_epoch=train_steps,
                             epochs=100,
                             validation_data=audio_gen.next_val(),
                             validation_steps=valid_steps,
                             callbacks=[checkpoint],
                             verbose=1)

```

声学模型网络最终的训练结果如【代码清单 8-25】所示。

**【代码清单 8-25】声学模型网络训练结果**

```

Epoch 1/20 101/101 [=====] - 145s - loss: 246.7676 - val_loss: 215.9699
Epoch 2/20 101/101 [=====] - 144s - loss: 187.5169 - val_loss: 179.8064
Epoch 3/20 101/101 [=====] - 152s - loss: 151.9236 - val_loss: 146.9361
Epoch 4/20 101/101 [=====] - 152s - loss: 133.9369 - val_loss: 135.4676
Epoch 5/20 101/101 [=====] - 148s - loss: 122.4069 - val_loss: 121.8893

```

.....



```
Epoch 16/20 101/101 [=====] - 150s - loss: 57.3928 - val_loss: 58.4264
Epoch 17/20 101/101 [=====] - 147s - loss: 53.5605 - val_loss: 54.9806
Epoch 18/20 101/101 [=====] - 148s - loss: 49.3938 - val_loss: 50.0497
Epoch 19/20 101/101 [=====] - 150s - loss: 46.2359 - val_loss: 48.1326
Epoch 20/20 101/101 [=====] - 150s - loss: 43.3314 - val_loss: 42.6372
```

## 5. 网络预测

作者在经过多次训练该声学模型后，其中训练结果最好的一次是增加了声学模型中的神经元 `node` 数，并相应地增加训练集中的数据量，最终使得训练的损失值不断下降，提高训练准确率，最终拿出训练结果最好的一个（训练损失值最小的一个）。在【代码清单 8-26】中给出了网络的预测和最终的预测结果。

### 【代码清单 8-26】声学模型预测结果

```
def int_sequence_to_text(int_sequence):
    """ 把输出的索引号转换到对应的字母 """
    return [index_map[c] for c in int_sequence]

>>> data_point = data_gen.normalize(data_gen.featurize(audio_path))
>>> prediction = gru_model.predict(np.expand_dims(data_point, axis=0))
>>> output_length = [input_to_softmax.output_length(data_point.shape[0])]
>>> pred_ints = (K.eval(K.ctc_decode(prediction, output_length)[0][0])+1).
flatten().tolist()

# 输出预测结果
>>> print('-'*80)
>>> print('True transcription:\n' + '\n' + transcr)
>>> print('-'*80)
>>> print('Predicted transcription:\n' + '\n' + ''.join(int_sequence_to_text(pred_ints)))
>>> print('-'*80)

# 最终预测结果
-----
True transcription:

[ and this plan was adopted too in order to extract from me a promise that i
would depart in peace ]
-----
Predicted transcription:

[ and this plan was adopted too in order to extract from me a promise that i
would depart in peace ]
-----
```



## 8.6.5 语言模型

声学模型对音频 [HELLO] 的预测输出可能为 [AEE\_LL\_L\_OOOO]、[HAA\_L\_LL\_OOOO]，甚至是 [HEE\_LL\_L\_UUUU]。首先，我们可以一定程度上根据重复规则进行替换，然后删除单词中的空白字符：

```
[ A E E _ L L _ L _ O O O O ] ==> [ A E _ L _ L _ O ] ==> [ A E L L O ]
[ H A A _ L _ L L _ O O O O ] ==> [ H A _ L _ L _ O ] ==> [ H A L L O ]
[ H E E _ L L _ L _ U U U U ] ==> [ H E _ L _ L _ U ] ==> [ H E L L U ]
```

通过上述转换，我们可以得到3个不同的输出预测结果，如 [AELLO]、[HALLO] 或者 [HELLU]。上述3个单词实际上发声都与 [HELLO] 类似，声学模型负责每一帧预测一个音素，因此会得到音素所对应的字母。例如我们说 [hello my chicken wings]，最终声学模型的预测可能是 [hallu my thicken wins]，显然声学模型的预测并不符合我们的期望。

在自动语音识别系统中，我们不仅需要声学模型，同时还需要语言模型。将声学模型得到基于音素发音的预测输出和基于语言文本数据库得到的语言模型进行结合，调整和修改声学模型的输出。

很明显在实际生活当中，我们很少写 [AELLO]、[HALLO] 或者 [HELLU]，而是更多地使用 [HELLO] 作为书面表达，因此会选择 [HELLO] 作为最终的结果输出。同理，对于 [hallu my thicken wins] 经过语言模型后的输出应为 [hello my chicken wings]。

## 8.6.6 语音识别的展望

(G. Saon et. al, 2016) 表示，基于现有的语音识别技术，在特定的应用场景中，语音识别系统已经初步达到或者接近人类的识别能力。随着深度学习的发展，目前语音识别系统基于海量的用户语音数据，训练得到的通用语音识别系统在限定场景中已经达到了实用化的水平。

例如在全球知名的语音识别 Switchboard 任务中，IBM 已经可以将语音识别错误率降低到 5.5%，而且最新的错误率已经低于人类的错误率，降低到了 4%。因此出现了近年来在手机端的输入法语音输入功能，以及各大互联网巨头相继推出的智能音箱。然而，这并没有代表语音识别完全达到或者超越人类的识别能力。

然而，这并没有代表语音识别真的完全达到或者超越人类的识别能力。

- 即使在背景有强噪声的干扰下，人类仍然可以将注意力集中在某一个人的





谈话上，目前语音识别系统仍然很难以去实现人类听觉系统的这种功能。

- 其次是自动语音识别系统的鲁棒性较差，对于机器语音识别，更换使用场景（不同的背景噪声、不同的方言和口音、不同的录音设备）都会对语音识别的准确率有一定的影响，甚至导致自动语音识别系统不可用。
- 服务器需要使用海量的数据（几十万乃至几百万小时标注好的语音数据），尝试着重新训练获得一个鲁棒性更好的声学模型时，人类却不需要这么多语音数据就可以得到更好的识别效果。
- 远场识别依然是一个具有挑战性的问题，目前远场语音识别的错误率是近场的两倍以上，甚至更高。

因此，合理有效地解决远场问题以及强噪声干扰情况下的语音识别是目前有待进一步研究的热点问题。现今的主流做法是将语音识别技术和麦克风阵列相结合（软硬件相结合）。前端通过阵列信号处理技术，将多通道语音增强；后端利用深度学习进行声学建模。当然，该方案也有待进一步优化，比如如何将阵列信号处理技术和深度学习方法相结合，利用阵列信号处理的知识指导深度神经网络的结构设计，从而直接从多通道语音信号中学习多通道语音增强方法，然后与后端声学模型联合优化。

近年来使用深度学习技术，使得自动语音识别系统能够做到端到端的训练，大大加快了工程的进步。结合海量的数据加以训练等高超的工程技巧，使自动语音识别领域的性能得到显著的提高。正像微软亚洲研究院的黄学东院士所说的“达到人类水平的对话语音识别，与其说是算法的胜利，不如说是工程的奇迹”。

目前大部分语音识别系统属于一个通用的系统框架，但是每个人的发音以及用词习惯都存在一定的差异性。如何使得语音识别系统像个人推荐系统一样，更加有针对性和智能化、更加懂“我”，将会是未来语音识别的研究热点！

## 8.7 本章小结

基本的循环神经网络结构模型存在着原始设计的不足，不能够快速有效地处理长期依赖的问题，限制了循环神经网络的记忆能力，于是我们深入地了解了 LSTM 和 GRU 网络架构学习如何避免该问题。另外我们还了解了基于传统的机器翻译的工作流程，然后使用循环神经网络的机器翻译建立了编码 - 解码网络模型，进一步对长期序列信息进行处理。基于 LSTM 的普通编码 - 解码网络模型依然存在着很多问题，会限制机器翻译的精度，于是出现了 Seq2Seq 网络模型的各种



种变体。

- LSTM 循环网络架构，通过输入门、遗忘门、输出门存储长期信息。
- GRU 是 LSTM 的精简版本，不仅简化了记忆计算的整体流程，还能够缩短训练的时间，对于处理长期依赖的序列数据是一个不错的选择。

## 引用/参考

- [1] Brown P F, Cocke J, Pietra S A D, et al. A statistical approach to machine translation[J]. Computational Linguistics, 1990, 16(2):79-85.
- [2] Brown P F, Pietra V J D, Pietra S A D, et al. The mathematics of statistical machine translation: parameter estimation[J]. Computational Linguistics, 1993, 19(2):263-311.
- [3] Al-Onaizan Y, Curin J, Jahr M, et al. Statistical Machine Translation[J]. Desidoc Journal of Library & Information Technology, 1999, 30(2):25-32.
- [4] Callison-Burch C, Talbot D, Osborne M. Statistical machine translation / [M]. Cambridge University Press, 2010.
- [5] Collobert R, Weston J. A unified architecture for natural language processing: deep neural networks with multitask learning[C]// International Conference on Machine Learning. ACM, 2008:160-167.
- [6] Sutskever I, Vinyals O, Le Q V. Sequence to Sequence Learning with Neural Networks[J]. 2014, 4:3104-3112.
- [7] Vinyals O, Le Q. A Neural Conversational Model[J]. Computer Science, 2015.
- [8] Cho K, Van Merriënboer B, Gulcehre C, et al. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation[J]. Computer Science, 2014.
- [9] Sutskever I, Vinyals O, Le Q V. Sequence to Sequence Learning with Neural Networks[J]. 2014, 4:3104-3112.
- [10] Bahdanau D, Cho K, Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate[J]. Computer Science, 2014.
- [11] Danescu-Niculescu-Mizil C, Lee L. Chameleons in imagined conversations: a new approach to understanding coordination of linguistic style in dialogs[C]// The Workshop on Cognitive Modeling and Computational Linguistics. Association for Computational Linguistics, 2011:76-87.
- [12] Cho K, Van Merriënboer B, Bahdanau D, et al. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches[J]. Computer Science, 2014.
- [13] Mozer S, M C, Hasselmo M. Reinforcement Learning: An Introduction[C]// IEEE Press, 2005:285-286.



- [14] Cho K, Van Merriënboer B, Bahdanau D, et al. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches[J]. Computer Science, 2014.
- [15] Gers F A, Schmidhuber J. Recurrent Nets that Time and Count[C]// Ieee-Inns-Enns International Joint Conference on Neural Networks. IEEE Computer Society, 2000:3189.
- [16] Amodei D, Anubhai R, Battenberg E, et al. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin[J]. Computer Science, 2015.



本书作为深度学习的入门实践教程，对人工智能的理论进行深度剖析，主要介绍了人工神经网络（ANN）、卷积神经网络（CNN）、循环神经网络（RNN）的相关内容，囊括了经典和前沿的深度学习框架算法（从数据分类、图像识别分类、图像风格迁移、语音识别到自然语言处理），对算法的核心思想进行讲解，并辅以大量图例和案例代码解析，理论结合实践，让读者领略深度学习的真正魅力。

将一本技术书籍写得通俗易懂谈何容易，但《深度学习原理与实践》这本书确实做到了。书中对近年来火热的深度学习理论知识进行简单剖析，化繁为简，没有局限于坐而论道，而是将实例和数学理论相结合，让读者能够快速理解各种模型并上手实践，值得细读。

——唐春明 广州大学数学与信息学学院副院长

本书从原理、方法、实践这3个维度系统地介绍了深度学习的方方面面，内容详实，解读清晰，细节与全貌兼顾，既适合初学者阅读，也可以作为深入研究的参考用书。

——杨刚 西安电子科技大学教授

近年来出版的深度学习相关图书中，本书是我见过非常有指导意义的中文书籍之一。本书对ANN、CNN、RNN等模型进行深入浅出的介绍，引入大量图例和简化后的公式，让算法浅显易懂。每一章的实践内容都给人惊喜，强烈推荐！

——吴健之 腾讯音乐高级工程师

作为产品经理，我能看懂的深度学习书籍实在太少了。本书恰到好处，插图丰富直观，数学公式简练，很喜欢此类风格的图书，易懂好学。即使你不是程序员或算法专家，该书也值得一看！

——张瑞 中软国际高级产品经理



异步社区 [www.epubit.com](http://www.epubit.com)  
新浪微博 @人邮异步社区  
投稿/反馈邮箱 [contact@epubit.com.cn](mailto:contact@epubit.com.cn)



ISBN 978-7-115-48367-6

定价：89.00 元

封面设计：广领设计

分类建议：计算机 / 深度学习

人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)



版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF